



**IMPLEMENTATION AND ANALYSIS OF THE
PARALLEL GENETIC RULE AND
CLASSIFIER CONSTRUCTION ENVIRONMENT**

THESIS

David M Strong, Captain, USAF

AFIT/GCS/ENG/01M-14

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

20010706 164

The views expressed in this thesis are those of the author and do not reflect
the official policy or position of the United States Air Force,
Department of Defense, or U. S. Government.

IMPLEMENTATION AND ANALYSIS OF THE PARALLEL GENETIC
RULE AND CLASSIFIER CONSTRUCTION ENVIRONMENT

THESIS

Presented to the Faculty of the Graduate School of Engineering
and Management of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

David Strong

Captain, USAF

March, 2001

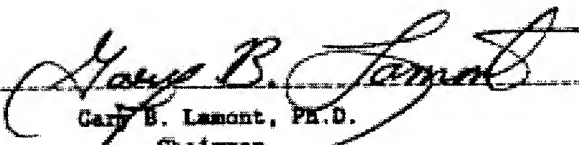
Approved for public release; distribution unlimited

IMPLEMENTATION AND ANALYSIS OF THE PARALLEL GENETIC
RULE AND CLASSIFIER CONSTRUCTION ENVIRONMENT

David M. Strong, B.S.E.E.


Captain, USAF

Approved:



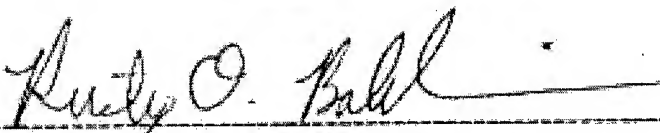
Gary B. Lamont, Ph.D.
Chairman

25 FEB 01
date



Eric P. Magee, Ph.D., Maj, USAF
Member

5 MAR 01
date



Rusty D. Baldwin, Ph.D., Maj, USAF
Member

5 Mar 01
date

Acknowledgments

First and foremost, I thank the Lord for the life, family, friends, and opportunities that I have been given. Second, my deepest appreciation and thanks go to my beautiful wife, . She graciously put up with many late nights and the subsequently less than cheerful days that followed. Finally, to the two most precious bundles of joy in my life: . They arrived two weeks before I started this program and have definitely made life more interesting.

David M Strong

Contents

Acknowledgements	v
Table of Contents	vi
List of Figures	ix
List of Tables	x
Abstract	xi
1 Introduction	1
2 Background	4
2.1 What is Data Mining?	4
2.1.1 Feature Selection	5
2.2 Genetic Rule and Classifier Construction Environment	7
2.2.1 Objective vs Fitness Structure	8
2.3 Parallel Processing	9
2.3.1 Control Parallelism vs. Data Parallelism	12
2.3.2 Measuring Performance	13
2.4 Summary	15
3 Design and Implementation	16
3.1 The Porting of GRaCCE	17
3.2 Serial Performance Analysis	18
3.2.1 Baseline Data Sets	19
3.3 Parallel Implementation	19
3.4 Testing the Parallel Implementation	20
3.5 Modifications for Heterogeneous Environments	20
3.6 Control of the Environment	21

3.7 Summary	22
4 Results and Analysis	23
4.1 Serial Performance Analysis	23
4.1.1 Serial Execution Profiling	24
4.1.2 Serial Timing Baseline	26
4.2 Parallel Implementation	27
4.3 Heterogeneous Environments	34
4.4 Summary	35
5 Conclusions and Recommendations	36
5.1 Summary	36
5.2 Conclusions	36
5.3 Recommendations	37
A AFIT Cluster of PCs	39
B Evolutionary Algorithms	40
B.1 Genetic Algorithms	41
B.2 Representations	42
B.3 Selection Operators	44
B.4 Search Operators	45
C GProf Flat Profile of GRaCCE FS	47
D GProf Call Graph of GRaCCE FS	51
E Main GRaCCE Module	56
F GRaCCE's Feature Selection Module	62
G GRaCCE Default Inputs	72

List of Figures

1	General Algorithm for Data Mining	1
2	Observation, Orientation, Decision, and Action Loop	5
3	Example of Samples in a Two Class Data Set	6
4	GRaCCE C++ Call Graph	9
5	Objective Landscape for the Iris Data Set	10
6	Partial Objective Landscape for the Wine Data Set	10
7	Example Fitness Landscape for the Iris Data Set	11
8	Serial GRaCCE Timing	25
9	Significant Profiler Lines for GRaCCE	26
10	Feature Selection Genetic Algorithm	28
11	Possible Parallel Objective Function Approaches	29
12	Iris Data Set Test Results	31
13	Wine Data Set Test Results	31
14	Glass Data Set Test Results	31
15	Cancer Data Set Test Results	32
16	Speedup	32
17	Efficiency	33
18	Iris with Multiple Populations	34
19	Effects of Heterogeneous Algorithm	35
20	Bäck's Basic Evolutionary Algorithm	40

List of Tables

1	Worst Case Time Complexity Analysis of GRaCCE	8
2	Serial Runtime of Iris Data Set	18
3	Sample Data Sets	19
4	Serial Execution Baselines by Processor Speed	27
5	AFIT's Pile of PCs	39

Abstract

This paper discusses the Genetic Rule and Classifier Construction Environment (GRaCCE), which is an alternative to existing decision rule induction (DRI) algorithms. GRaCCE is a multi-phase algorithm which uses evolutionary search to mine classification rules from data. The current implementation uses a genetic algorithm based 0/1 search to reduce the number of features to a minimal set of features that make the most significant contributions to the classification of the input data set. This feature selection increases the efficiency of the rule induction algorithm that follows. However, feature selection is shown to account for more than 98 percent of the total execution time of GRaCCE on the tested data sets. The primary objective of this research effort is to improve the overall performance of GRaCCE through the application of parallel computing methods to the feature selection algorithm. The development and implementation of a parallel feature selection algorithm is presented. The experiments designed and used to test this parallel implementation are outlined followed by an analysis of the results. The results of this thesis effort show clearly that GRaCCE is improved through the use of parallel programming techniques.

IMPLEMENTATION AND ANALYSIS OF THE PARALLEL GENETIC RULE AND CLASSIFIER CONSTRUCTION ENVIRONMENT

1 Introduction

As computers get faster and faster they are used to generate more and more data for processing. Automated tools are needed that can help in understanding these large volumes of data. One area of automated processing of this voluminous data is called data mining. Similar to mining for gold in a mountain, data mining is looking for those nuggets of information that make better sense of the data. Techniques in data mining are being developed for use in many areas such as pattern recognition, associations, change and anomaly detection, among others. Without some way of interpreting the vast amounts of data, it is nearly useless to try due to lengthy computation time. Data mining tools are developed to help in this interpretation and attempt to speed up the lengthy computations. The general steps in a data mining algorithm are shown in Figure 1.

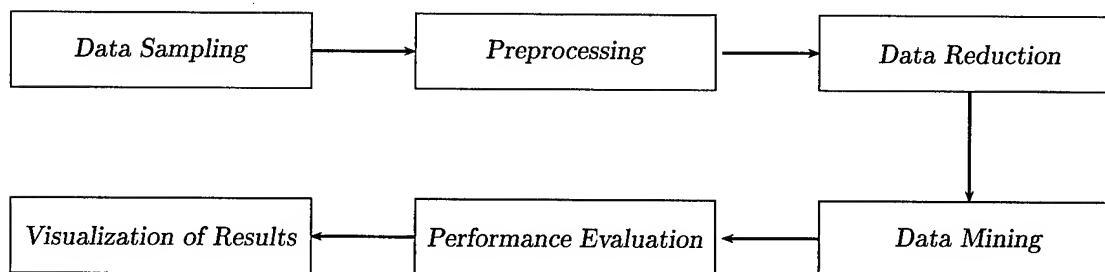


Figure 1: General Algorithm for Data Mining

Due to the overwhelming size of some databases today, it is impractical to process them in their entirety. Thus, some way of reducing the amount of information processed while attempting to still maintain similar results is important. This process is called *feature selection*. In the pattern recognition area, feature selection attempts to remove features from consideration that make little or no contributions to distinguishing between classes in the data set.

Once data is selected for processing, a preprocessing routine may be required to translate the

data into a usable form for data mining or even to repair missing data. Care should be taken in converting data from one context to another so that the impact from any added errors on later results are as small as possible. Data reduction techniques may now be used to reduce the data to a minimal set of features that directly contribute to classifications.

The data mining technique is applied to this minimal feature set to produce the classifier rule set. Once a rule set has been developed, it can be evaluated based on criteria such as percent correct classifications versus mis-classifications. This testing should use samples that weren't used in the generation of the rule set to check whether or not valid rules are generated. Finally, some understandable presentation of these rules should be made to the user.

This thesis effort investigates one data mining tool and the use of parallel processing to improve its performance. The tool used is the Genetic Rule and Classifier Construction Environment (GRaCCE) developed by Marmelstein [20] and is discussed in Section 2.2. The primary emphasis is to analyze GRaCCE and re-implement portions of it in a parallel environment to increase its speed of execution. The overall objectives of this thesis effort are as follows:

1. Port GRaCCE to run on the Linux [29] operating system,
2. Analyze serial GRaCCE's efficiency and effectiveness for bottlenecks,
3. Re-implement any bottlenecked sections to improve efficiency, and
4. Analyze the parallel code effects on performance.

The scope of this effort is limited to improvements in the computational performance relative to wall clock time by parallel implementation of selected sections in GRaCCE. Other improvements to the overall GRaCCE algorithm are beyond the scope of this effort and are not researched. The approach taken is to analyze the serial GRaCCE code for sections that are inefficient and develop a parallel design to improve the sections efficiency. The resulting thesis is outlined in the following section.

Outline of Thesis

The thesis begins with a general introduction of the topic areas involved followed by chapter two which provides more specific background information on the main topic areas. The first section is on data mining followed by information on the specific data mining tool used in this effort. The next sections are on serial and parallel computing issues and performance evaluation metrics. Chapter three discusses the methods used in the analysis of the serial code, the development of the parallel code, and the testing required for analysis of any improvements. Chapter four presents the data collected from the serial analysis to the parallel execution. Additionally, the effects of the parallel implementation on performance is analysed versus the serial implementation. In chapter five conclusions are made regarding any parallel improvements and recommendations for future efforts.

2 Background

With today's faster computers and larger hard drives, the amount of information available for processing is continuously growing. A great deal of time and resources have gone into building databases. As a database increases in size, however, it becomes increasingly difficult for an individual to comprehend all of the data. For larger databases, some tools are needed to separate the gold from the dross, in order to utilize the database to the greatest extent. Like mining for gold, data mining is an approach used in the search for gold nuggets of information within the mountains of data. This chapter discusses data mining, some methods used in data mining implementations, and methods of improving data mining performance through parallel processing.

2.1 What is Data Mining?

Many of today's very large databases are not very useful unless the data can be grouped into meaningful sub-sets [14]. This partitioning of the data in the database is known as classification. The particular form of classification considered in this research is *data mining*. Data mining is a broad term used to describe any process that seeks to uncover patterns, associations, changes, anomalies, or statistically significant partitions in data [20]. Traditional data analysis is performed manually by developing a hypothesis and then testing to see if the data supports it. In contrast, data mining is an automatic process that discovers useful patterns in the data and extracts them [12].

With the prolific use of databases by many businesses, many data mining efforts have a distinctly commercial orientation [1]. However, there are many possible military applications with the growth in the number and size of military databases. One such application is the augmentation of the *Observation, Orientation, Decision, and Action* (OODA) loop shown in Figure 2. The OODA loop is one of the most widely accepted models of the battlefield decision process [27]. Today's battlefields are full of various data collection devices that collect vast amounts of information. This information is gathered for processing with the hope that relevant information is presented to the decision makers. Due to the large amounts of data and the timeliness required, data mining is a good addition to the OODA orientation process. The many sensors gather the data which gets stored in a database during the observation process. Next, data mining is applied to the collected

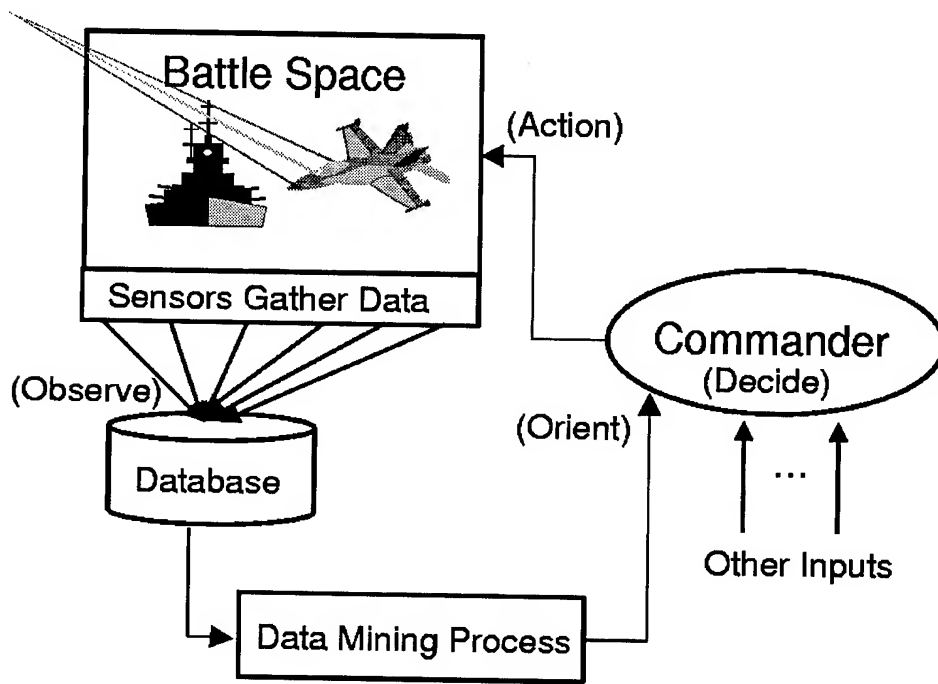


Figure 2: Observation, Orientation, Decision, and Action Loop

data to detect emerging patterns or features that weren't previously known. These patterns can be used to improve the data content provided to the commander for use in the decision process. One improvement may be decreased processing time if data mining results show that built in classifiers can have the same (or better) performance with a reduced feature set. This reduction in feature set size is called feature selection.

2.1.1 Feature Selection

Selecting features is an extremely difficult task, filled with both theoretical and computational problems. An effective mathematical theory for feature selection seems achievable only for a narrow specialization of the problem: linear transformations that reduce the dimensionality of the feature space. The transformation is made with the assumption that data is drawn from a normal distribution [24].

From the standpoint of Bayesian decision rules, there are no bad features. One cannot improve the performance of a Bayes classifier by eliminating a feature, a property called monotonicity. How-

ever, in practice the assumptions in the design of Bayes classifiers are almost never valid. As a consequence, it is possible to improve the performance of a non-ideal classifier by deleting a feature. Moreover, for a given amount of data, reducing the number of features increases the accuracy of the classifier's performance [24]. These two facts have tremendous consequences for computational problems associated with feature selection.

While features within a data set provide the means for discriminating between classes [16], too many features can degrade a system's ability to classify data [5]. One reason is the number of samples required for training increases as the number of features and possible values for these features increase, a phenomenon termed the "curse of dimensionality" [20, 4]. For a data set with d features with M possible values for each feature, M^d samples are required for training to truly be effective. Additionally, not all features are effective in discriminating between classes. An example data set with two classes is shown in Figure 3.

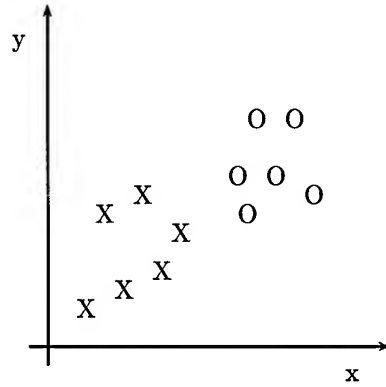


Figure 3: Example of Samples in a Two Class Data Set

One way to fight the curse of dimensionality is to reduce the feature set size. Given a data set with d features and a desired subset of m features, how is the best subset found? An exhaustive search would have to try $\binom{d}{m} = \frac{d!}{(d-m)!m!}$ possibilities [28]. For example, given a data set with thirty-five features and the best twenty are desired, an exhaustive search would have to test 183,579,396 possible combinations.

Due to the prohibitive time an exhaustive search would take, other methods of searching for

feature subsets are required. Several approaches such as the Fischer Discriminant [9] (FD), feature extraction using the *Karhunen-Loeve* (KL) transformation, and heuristic search using evolutionary techniques are discussed in [20] in greater detail. The FD calculates the impact of the individual features on the classification but does not allow for combinations of features. The KL transformation assumes that some function can be developed to transform the existing data set into another easier to classify data set. The third method is to use a heuristic search such as genetic algorithms to find a solution and is the method used by the software in this thesis effort.

2.2 Genetic Rule and Classifier Construction Environment

GRaCCE is a data mining tool that uses evolutionary search techniques[19] to mine classification rules from the data it is given. It is similar to a pattern recognition algorithm, but goes beyond by producing understandable rules used in the recognition. GRaCCE is designed for use with samples in a vector form [20]. The individual elements in the vector are independent features which may be real-valued or discrete. Getting data into this form is the primary goal of the first step in the data mining process.

Once the data is in the proper form, GRaCCE can begin with a pre-processing phase that includes an optional feature selection operation and/or a mandatory *winnowing* operation. Winnowing removes statistical outliers and duplicate samples resulting in classes of data that are linearly separable. GRaCCE then selects the relevant data sets for training through either a random sampling of data or a cross-validation method. Next, it executes a data mining algorithm that can be viewed as a combination of clustering and classification, since it forms a classification model of data based on data attributes that include the clustering of class homogeneous regions. The steps that the data mining algorithm follow sequentially are: partition generation, data set approximation, region identification, region refinement, and partition simplification. Worst case analysis by section is shown in Table 1, where n is the number of samples and d is the number of features. The results of the data mining algorithm are finally interpreted and evaluated through a listing of the partitions that form these regions in the data. The overall process flow of GRaCCE is shown in Figure 4.

Phase	Complexity
Pre-processing - Feature Selection	$O(n^4)$
Pre-processing - Winnowing	$O(dn^2d)$
Partition Generation	$O(2(dn)^2 + nd)$
Data Set Approximation	$O(n\log(n))$
Region Identification	$O(qn^2d^{3/2})$
Region Refinement	$O(dn\log(n))$
Partition Simplification	$O(d^2n^2\log(n))$

Table 1: Worst Case Time Complexity Analysis of GRaCCE

At the top level is the root GRaCCE process *gracce*. GRaCCE has three main modules at the second level along with the *load* module. The load module is used to read in the test data set prior to any of the other modules being called. The first of the three main modules is the feature selection module. The other two modules are *winnow* and *ruleInd*. The rule induction module makes the most extensive use of the rest of the GRaCCE code and calls the five modules at the base level in Figure 4. The *ga* module provides method for both the feature selection routines along with the region ID section of the rule induction phase.

2.2.1 Objective vs Fitness Structure

Most evolutionary computation (EC) algorithms require some form of encoding or mapping [3] between the problem domain and the algorithm domain. The evolutionary operators, discussed later, operate on these encoded structures sometimes called chromosomes. GRaCCE uses a binary string representation for each population member with a length determined by the number of features in the data set. Each feature is represented by a single binary digit. A one represents the feature is present and a zero represents the feature is absent.

The fitness landscape [20] for an EC algorithm refers to the mapping between the genomes of a population of individuals to their fitness and a graphical representation of that mapping. For GRaCCE, each population member is evaluated using the objective function in the feature selection code. The objective function performs a k-nearest neighbor (kNN) routine over a subset of the data for each member in the population. The subset is determined by the bit representation of the individual member. The complete objective function landscape for the Iris data set [6] is shown in

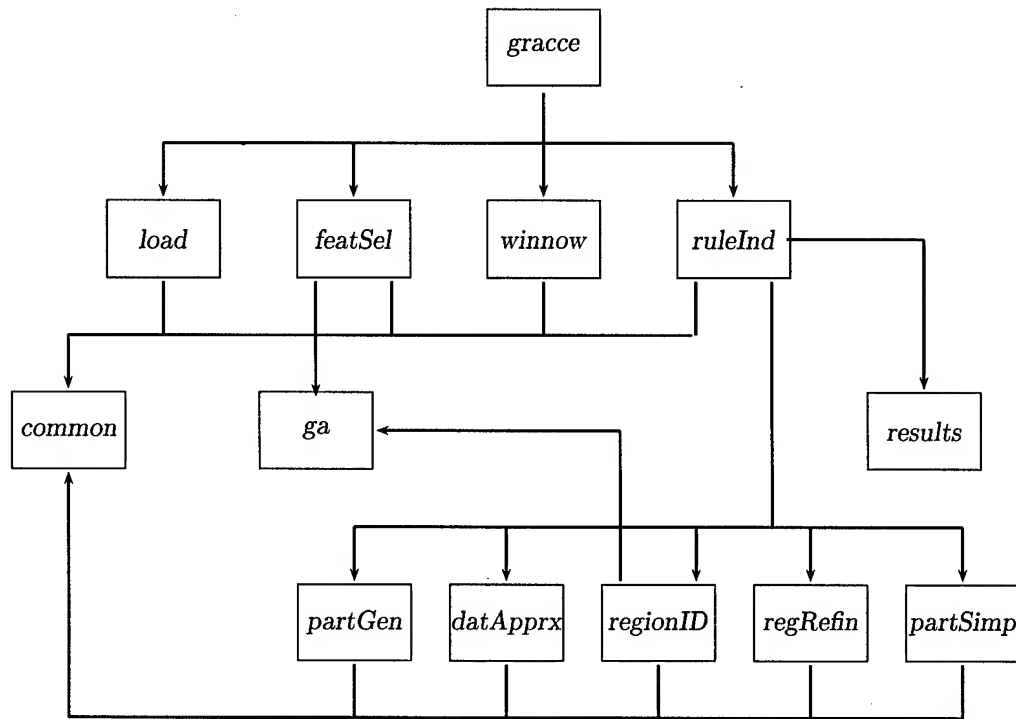


Figure 4: GRaCCE C++ Call Graph

Figure 5. A partial objective landscape for the Wine data set is shown in Figure 6.

GRaCCE's fitness function receives the results of the objective function and then ranks the members based on those results. The fitness function produces a rank ranging from zero to two based on each members objective score relative to the whole population. The objective score for a particular member does not change for the entire execution of the genetic algorithm (GA) as long as its binary string remains unchanged. However, the fitness of that member is recalculated and may change with each generation. An example of the fitness mapping is shown in Figure 7. This ranking was produced using the second generation of the feature selection GA in GRaCCE with a population size of forty.

2.3 Parallel Processing

Imagine a post office in a large city with only one person responsible for processing all of the incoming mail. This person takes stacks of mail and walks around to the different bins. The person has to

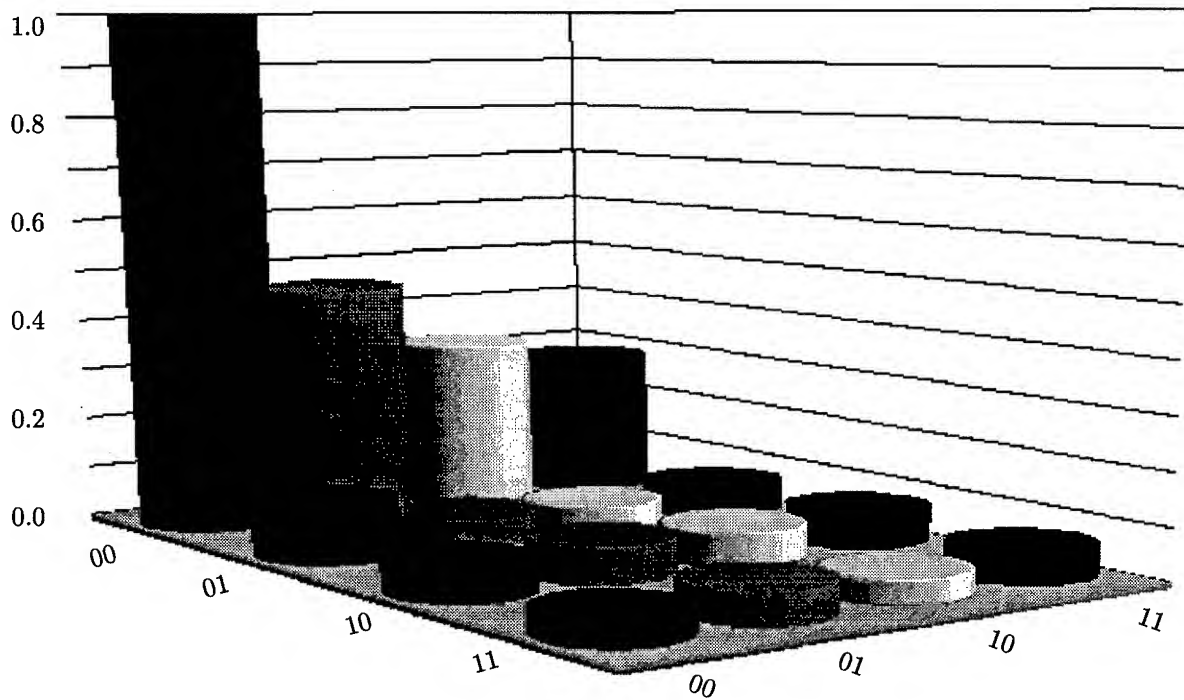


Figure 5: Objective Landscape for the Iris Data Set

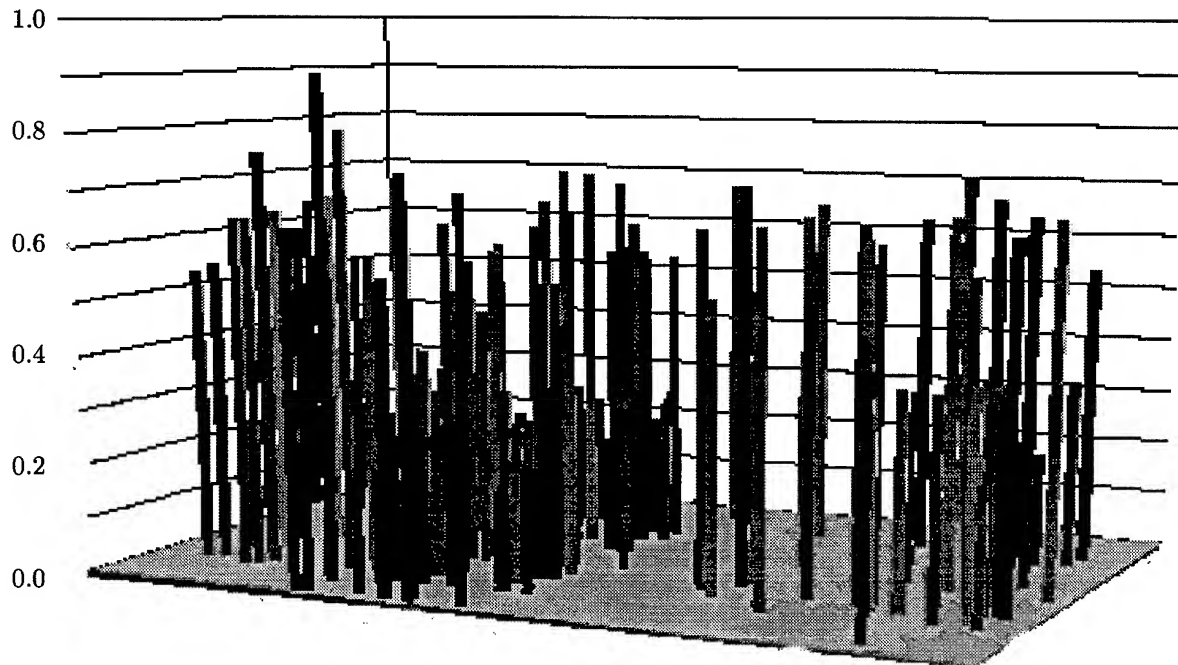


Figure 6: Partial Objective Landscape for the Wine Data Set

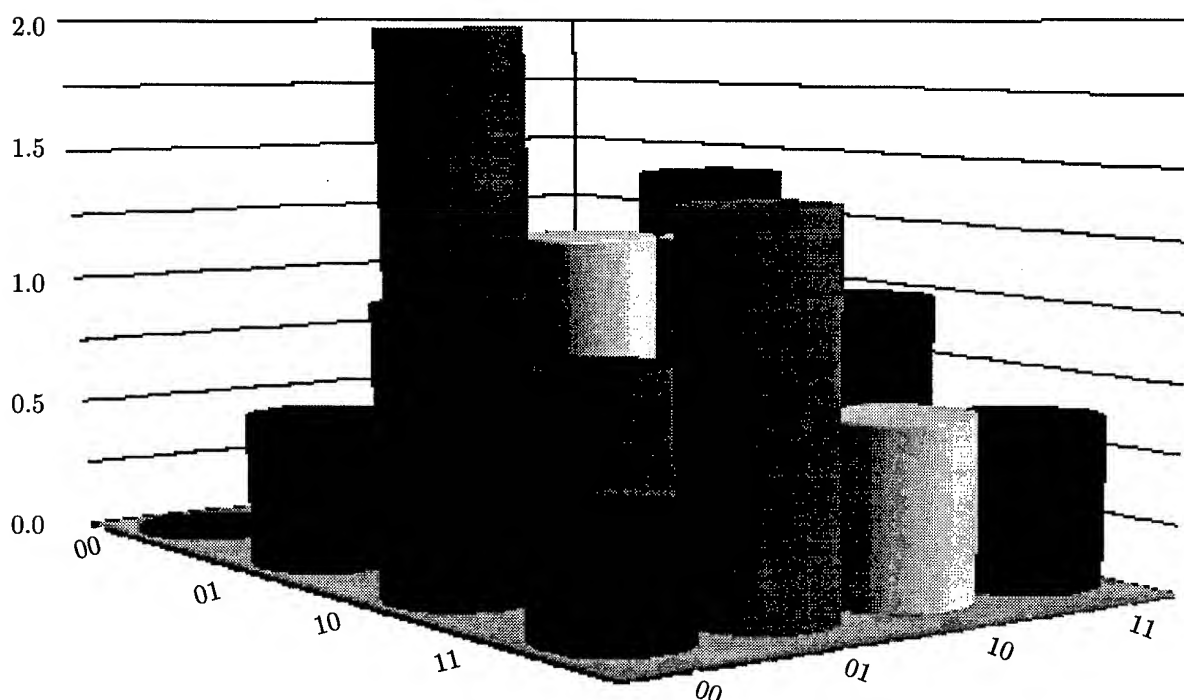


Figure 7: Example Fitness Landscape for the Iris Data Set

sort the mail into the bins by ZIP code. A single person can only do this so fast. One way to speed this up is to add another person to help. Working together they can sort the mail twice as fast. If two more people are added, the mail is sorted at a rate four times as fast as a single person.

In addition to the rate at which the task is accomplished, other factors such as efficiency can affect overall performance. In the previous case, all the workers are walking around to all of the bins. A more efficient method of splitting up the workload can be used. The total number of bins can be divided among the workers. Each worker is then responsible for a smaller number of bins and takes a portion of the incoming mail. This is an example of *task partitioning* or a *divide-and-conquer* approach. The worker files the mail that matches the workers' bins and passes non-matching mail to the worker responsible for it. This cuts down on the amount of walking around that each worker does.

This may, however, increase the efficiency only to a certain point. If the workers are required to verbally communicate when passing along mail to the responsible worker, this adds some communications overhead. As the number of workers increases, the communication burden gets larger. At

some point, the communication requirement starts to dominate a workers' time. Adding even more workers only makes the situation worse and performance can only decrease. If workers are added beyond even this point, it could take them longer to sort the mail than it did for a single worker.

The mailroom analogy is similar to many computer programs. Computer programs often perform the same task many times. Instead of adding workers, additional processors or computers can be added to increase performance. For instance, the mailroom case is analogous to sorting routines which vary in speed and efficiency and sometimes benefit from the addition of processors in parallel. Two such parallel sorting algorithms[26] are:

1. **Odd-Even Transposition** with a worst case time of $O(n)$ and
2. **Shear Sort** with a worst case time of $O(n^{1/2} \log n)$.

How much performance increases is determined primarily by the amount of communications required between processes. Two important measures of how well a program can be executed in parallel are speed up and efficiency.

2.3.1 Control Parallelism vs. Data Parallelism

Two major categories of parallel programming are data-parallel programming and control-parallel programming[23]. Control-parallel algorithms can apply different operations to different data concurrently. This is considered a pipelined approach and the data flow in this case can vary widely in complexity. Any manufacturing assembly line is a good example of this type of parallelism. On the other hand, data-parallel algorithms split the data among the different nodes and apply the same operations to that data. The operations on the data are typically independent and the communication between nodes is much simpler than control-parallel algorithms. These differences in operation and structure between parallel algorithms and their serial equivalents makes performance much more difficult to measure.

2.3.2 Measuring Performance

Parallel programs can't be evaluated in the same simple terms, such as execution time and input size, used for a serial program. This is due to additional issues such as the use of multiple processors and different communication architectures. The combination of a program written for parallel execution and a parallel architecture is known as a parallel system [18]. Once a program has been developed or converted to run in a parallel environment, it must be tested using metrics that have meaning for a parallel system.

Several metrics have been developed for parallel systems with the most common being the parallel run-time (T_p). T_p is measured from the start of the program to the end of the last process. T_p is used in the computation of other common parallel metrics such as *speedup*, *efficiency*, and *isoefficiency*. These three metrics are discussed in the next three sections.

Speedup

Speedup is defined as “the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors. [18]” The speedup equation is

$$Speedup(S) = \frac{T_s}{T_p} \quad (1)$$

where T_s is the best serial execution time and T_p is the parallel execution time. This metric provides a basic measure of the gain in performance by parallelizing the algorithm.

In theory, the best speedup that can be obtained is a *linear* speedup [18]. Ideally, if a program runs in time T_s on a single processor then it should run in time T_s/p on p processors. In reality, this can never be achieved due to communications overhead required in the distribution of multiple processes. A deceptive condition known as *super-linear* speedup occurs in which the observed parallel speedup is greater than T_s/p . Unless there are no other constraints such as the serial algorithm is memory bound, this really implies that the parallel algorithm is encoded more efficiently and that

the serial algorithm could be recoded to match. By itself, speedup doesn't provide a completely accurate portrayal of the algorithm's performance.

Efficiency

A better understanding of the algorithm's performance is achieved by looking at the parallel algorithm's efficiency in addition to speedup. Although the speedup for a program may continue to increase as processors are added this will eventually taper off. This is a result of the parallel program not getting one hundred percent of the processor's time as each node is added. This small percentage of other tasks taking processor time adds up as nodes are added and causes the speedup to fall off. The efficiency function, E , is

$$E = \frac{S}{p} = \frac{T_s}{pT_p} \quad (2)$$

where S is the speedup and p is the number of processors. This metric measures the benefit of adding more processors.

Due to the previously mentioned upper bound, p , of speedup, the efficiency of a parallel program will never exceed one. If the efficiency can be maintained as processors are added and the program is scaled accordingly, the program is said to be *scalable*.

Isoefficiency

The rate at which the problem size has to be scaled as the processors are added to maintain a fixed (iso) efficiency is calculated by the *isoefficiency* function is

$$W = \frac{E}{1 - E} T_o(W, p) \quad (3)$$

where W is the workload or problem size, E is the efficiency, and T_o is the parallel overhead. T_o is a function of the problem size and the number of processors used. The idea is to maintain the same (iso) efficiency as the number of processors changes.

The main difference between the efficiency function and the isoefficiency function is that the overhead of the parallelization is taken into account. The lower the value of the isoefficiency function the more scalable the program is. If however, a fixed efficiency cannot be maintained regardless of how fast the problem is scaled, the program is considered *unscalable*[18].

2.4 Summary

In this chapter, data mining, GRaCCE, and parallel processing were discussed. Data mining is important to the extraction of valuable information from increasingly large databases. The data mining tool used in this thesis effort is GRaCCE and like other data mining tools, it might benefit from a parallel implementation. Finally, some considerations and characteristics of parallel programming are discussed to provide background in parallel processing. The next chapter discusses design and implementation issues that impact the development of a faster version of GRaCCE using parallel computing.

3 Design and Implementation

The main emphasis of this thesis effort is to improve the performance of the GRaCCE algorithm through use of parallel computing. The first steps in developing a parallel application are deciding on what tools, languages, and computing platforms are used. Once these decisions are made, developing the actual code commences. Given that this effort was provided a serial implementation of GRaCCE, a parallel version was not built from scratch, but made use of the serial code.

With these steps in mind, the first design decision is easy. Having received the GRaCCE code in C++ language, C++ is used since porting GRaCCE to another language is beyond the scope of this effort and is probably a separate thesis in itself. The next design decision is based on computing platform availability and control of the test environment. The Air Force Institute of Technology's (AFIT) PC cluster is isolated from the outside world and is the best system available for development and control of the parallel application test environment, therefore, it is chosen as the development platform.

The choice of AFIT's PC cluster comes with some constraints. The first of which is the hardware architecture, which is a cluster [7] of commodity PCs connected by an Ethernet communications bus (see Appendix A. The second constraint is the operating system. The individual nodes in AFIT's cluster are dual bootable under either Microsoft Windows 2000 or the Linux operating system. The Linux operating system is chosen over Windows 2000 based on prior experience with Linux and the cluster's history of greater stability and scalability under Linux.

Having made these choices the next steps are the analysis of the serial code, development of a parallel approach, implementation of the parallel code, running a set of experiments on the parallel implementation, and the analysis of the results. Within these steps it is important that the design of the experiments and the resulting tests are made in accordance with the basic goals of the thesis effort [8]. Along with identifying the desired goals, the means required to achieve them should also be identified. Additionally, any analysis of the experimental results must be sufficient to justify the claims made regarding the outcome [15]. Finally, any parameters that might influence the testing should be controlled as best as possible.

In this chapter several implementation and testing issues are discussed in the following sections. The first is the porting of the original C++ code to run on the Linux OS. Next is the performance analysis of the serial GRaCCE program and the methods used in the analysis. The parallel implementation of the largest bottleneck in the GRaCCE code is third, followed by the test methodology used to gather the data for analysis of this parallel implementation.

3.1 The Porting of GRaCCE

GRaCCE was originally developed by Marmelstein [20] using the Matlab environment. One drawback of Matlab is the slow speed at which the code executes. For this reason, Kinzig [16] ported GRaCCE from Matlab into C++ using Microsoft's Visual C++ environment and staying as close to the original Matlab code structure as possible. The idea is to provide C++ code structure that is easy to follow for users who understand the Matlab version.

In order to port the Matlab code to C++ and still maintain similarity in matrix manipulation a matrix library is used. This library is the Template Numerical Toolkit (TNT) [21] and is built using C++ templates. TNT was developed and tested under Microsoft's Visual C++. Two additional libraries are required for compilation of GRaCCE under either Windows or Linux and they are:

1. **BLAS** - a basic linear algebra library and
2. **LAPACK** - a library of Fortran 77 subroutines for solving some of the most common linear algebra problems and is an extension of the BLAS.

Version 3.0-8 of both BLAS and LAPACK are used to develop the Linux version of GRaCCE.

To fully utilize the nodes in AFIT's PC cluster, Kinzig's C++ code is ported to run under Linux and the GNU compiler suite. One element employed by the TNT library that is unsupported by the current stable release of the GNU C++ compiler is the passing of a function in the parameter list of the *map* procedure call. To correct this problem, separate individual procedures are written for each of the different functions. The GNU (recursive acronym meaning "GNU's Not Unix") compiler suite is used in the development of the Linux port of the C++ version of GRaCCE. The version of the GNU compiler suite on the AFIT cluster is *egcs-2.91.66*. As an alternative, the Portland

Group (PG) compiler suite is also available on the AFIT cluster for code development. However, the decision to use the GNU suite over the PG suite is based on familiarity with the GNU suite and because the GNU compiler reported fewer warnings and errors in the first attempts to port GRaCCE to Linux.

3.2 Serial Performance Analysis

Once the port to Linux is complete, its performance is analyzed. The code itself has timing measurements within it. GRaCCE makes system calls to the *clock()* subroutine at the start of a routine and again at the end of a routine. The difference is noted in seconds as the time required to execute the routine. These measurements are discussed in greater detail in [16] and are used to show the length of time required to execute the various sections of the GRaCCE code. The section requiring the greatest time, 98.9% of total execution time based on the Iris data set results shown in Table 2 is the feature selection routine. Due to the significant amount of time required for feature selection, it was chosen as the code segment to be implemented in parallel.

Iris Data Set on a 266MHz Pentium II			
	Mean	Std Dev	Percentage
Feature Selection	289s	6.3s	98.9
Winnowing	0.63s	0.01s	.22
Rule Induction	2.56s	0.09s	.88
Total	292.19		

Table 2: Serial Runtime of Iris Data Set

In order to get a better understanding of where the main bottleneck occurs in the feature selection routine, a profiling tool is used. Most compilers have profiling tools associated with them and are linked closely together. Profilers insert extra code into a program when it is compiled. This extra code generates a log file for a post mortem analysis. In this case, the GNU compiler suite comes with a profiler called *gprof*. Profiling is invoked by adding a command line parameter to the compiler invocation used to compile the code. The additional command is *-gp* and is placed anywhere after the *gcc* or *g++* compiler command.

3.2.1 Baseline Data Sets

A number of data sets are available for testing of classifier systems and a sampling of some of them are shown in Table 3. The Iris, Wine, Glass and Cancer data sets are chosen based on different sample and feature sizes. These data sets are timed using the serial version of GRaCCE on the different processor speeds available in the AFIT cluster. This establishes the baseline for analyzing any improvements made by parallelizing the GRaCCE code.

Data Set	Number of		
	Samples	Classes	Features
Iris	150	3	4
Wine	178	3	13
Glass	214	6	9
Soybean	683	19	35
Cancer	699	2	9
Iono	351	2	34
Pima	768	2	8
Scud	1000	2	6
Mushroom	8124	2	22

Table 3: Sample Data Sets

3.3 Parallel Implementation

Once the serial bottlenecks are determined, the modifications to the serial code with the greatest performance impact is implemented in parallel. Several tools are available for implementing GRaCCE in a parallel environment. Some of these possibilities[22] are the parallel virtual machine (PVM), high performance FORTRAN (HPF), the Local Area Multicomputer (LAM), and the message passing interface (MPI).

The idea behind PVM is to provide a unified framework within which parallel applications can be developed in an efficient and straightforward manner using existing hardware. Ideally, a collection of heterogeneous computers looks like one big parallel virtual machine[11]. PVM takes care of all the message passing, task scheduling and other tasks required to maintain the parallel environment without getting the user involved in the details.

LAM stands for local area machine and implements a lower level abstraction of the interconnec-

tions between the individual nodes than PVM. The user is much more aware that the environment is composed of multiple computers. The communications mechanism underlying LAM uses the message passing interface (MPI) standard [10].

The libraries used to implement the parallel GRaCCE code is MPICH version 1.2.0 which is a MPI standard implementation by Argonne National Laboratories. In order to ensure comparable results, the testing of this modification uses the same data sets used by the serial code. Additionally, program traces of execution of both the serial and the parallel algorithms are checked against each other to ensure proper execution of the parallel algorithm.

3.4 Testing the Parallel Implementation

After the parallel version of GRaCCE is implemented, several experiments are run. The first is to validate the correct execution of GRaCCE. This is accomplished by ensuring the output of both the serial and the parallel versions are the same. It is important to check not only the final reduced feature set that is produced but also proper execution throughout the entire test run.

Once the correct execution is validated, the parallel implementation is tested and analysed using the metrics discussed in section 2.3.2. This is accomplished by executing the parallel implementation on a multiple numbers of nodes. For this effort, the number of nodes used are two, three, four, five, ten, fifteen, and twenty. This number of nodes provides enough data points for analyzing the performance differences between the serial and the parallel implementations. Each set of runs are executed multiple times in the same configuration to allow for a statistical analysis of the results such as the mean execution time and the standard deviation of the results. The results of these tests are also analysed using the metrics discussed previously in section 2.3.2.

3.5 Modifications for Heterogeneous Environments

The initial parallel implementation of GRaCCE assumes a homogeneous cluster environment. This allows parallelization of the code, but overall performance is as if all of the nodes are the same as the slowest node used. Due to the heterogeneous nature of the AFIT cluster, additional modifications

are made to gain some performance improvements over the homogeneous implementation.

In order to take the heterogeneous nature of AFIT's cluster into account during program execution, some method of detecting system parameters is required. In this case, each node accesses the `/proc/cpuinfo` file which contains system specific information regarding the nodes processor. The file is searched for the processors *bogomips* rating. This number is directly related to the processor's clock speed and is easily accessible.

Other measures of performance such as those provided by HINT [13] are available. HINT is required to run *a priori* and produces a QUIPS rating which stands for quality instructions per second. The overall rating is based on a suite of tests that report performance related to CPU speed, cache size and speed, and bus speed, among other parameters. The QUIPS rating can be stored in a lookup table for a node to check what its rating is. ATLAS, on the other hand, stands for *automatically tuned linear algebra system* and continually updates its system performance value as it is executed. Both of these metrics are more tailored to systems that are more fully loaded in both computational and memory requirements. Due to GRaCCE's minimal memory requirements, the Bogomips rating is sufficient since it is more directly related to processor speed than the overall performance of a fully loaded system.

3.6 Control of the Environment

The AFIT cluster operates in an isolated environment. No outside connections to other networks are currently permitted so interference from outside network traffic is nonexistent. The network communications required by the parallel implementation of GRaCCE is very minimal. The initial distribution of the complete data set is the most time consuming and is considered negligible when compared to the overall GRaCCE feature selection time. All other communications are small $n \times m$ byte vectors between the root and other nodes where n is the size of the population in the GA and m is the number of features in the data set. Due to the small communications requirements, other parallel applications were permitted concurrent execution on the cluster with little effect on the timing results for GRaCCE.

Additional care is taken to ensure a minimal number of operating system processes are running in the background. None of the nodes used in timing GRaCCE have graphical windowing environments running during testing. The only other system processes enabled are those required for cluster operation such as the network file system (NFS) which provides the remote file system between nodes.

3.7 Summary

In this chapter, design and implementation issues concerning the development of the parallel GRaCCE code are discussed along with tests required to produce the data for analysis of the resulting code. The porting of the Microsoft Visual C++ version of GRaCCE to compile under the GNU compiler suite is at the beginning of the chapter. This is followed by the methods that are used to analyze the serial version of GRaCCE once ported. Next, some parallel implementation issues affecting a parallel version of GRaCCE are discussed, such as parallel environments and how to test the resulting parallel code. The chapter is concluded with a discussion of the controlled PC cluster environment used in development and testing of the parallel GRaCCE. The next chapter covers the choices actually made, implemented, and the results.

4 Results and Analysis

This chapter presents the design of the experiments along with the results and analysis of those experiments. The design of experiments entails stating the objectives of the experiments, identifying any influential factors on the experiments, what specific characteristics are to be measured and how, as well as the number of repetitions of the experiments [17].

Once the experiments are made, an objective of this chapter is to present the data gathered in a manner that is easily understood. The initial results are for the serial analysis of the GRaCCE port to Linux. This is followed by the implementation decisions made and why. The data collected based on that implementation is then presented. Next is an analysis of the parallel metrics based on the test results. The chapter is concluded by a discussion of the difference in performance between the homogeneous approach and a simple heterogeneous approach.

4.1 Serial Performance Analysis

Once the port to Linux is complete its performance is analyzed. The code itself has some timing measurements written within it and these measurements are discussed in greater detail in [16]. For an initial analysis, these measurements are used to show the length of time required to execute the various sections of the GRaCCE code. The timing measurements are made based on GRaCCE's menu which has four possible selections, as follows:

1. Winnowing Phase,
2. Feature Selection,
3. Rule Induction Phase, and
4. Exit.

These choices correspond to the three main modules shown in Figure 4. The *Winnowing Phase* winnows the original data set by removing duplicate entries and entries with missing data. The *Feature Selection* (FS) routine is set to use the GA method of feature selection. The other two methods, which are a depth first search (DFS) and a hybrid DFS-GA approach, were not used in

this analysis. The GA method for FS is used because it is the most time consuming of the three methods[16].

The *Rule Induction Phase* consists of the following six algorithms, each of which are individually timed:

1. Dividing Data,
2. Partition Generation,
3. Data Approximation,
4. Region Identification,
5. Region Refinement, and
6. Partition Simplification.

The individual times are summed together for a total rule induction phase time.

Based on the three GRaCCE menu selections, the section requiring the greatest amount of the total execution time is feature selection. As can be seen in Figure 8, the feature selection dominates the overall GRaCCE execution time. In fact, the timing of GRaCCE execution using the Iris data set resulted in the feature selection taking ninety-nine percent of the total execution time for all three parts.

With the significant amount of time required for feature selection, it was chosen as the target for further analysis.

4.1.1 Serial Execution Profiling

Once the feature selection part of GRaCCE is chosen, more detailed analysis of the code is required to narrow down the location of the major bottlenecks. For this, the GNU profiling tool *gprof* is used. In order to use *gprof*, GRaCCE is recompiled with the compiler flag '-pg' added to the compiler command line. This inserts the profiling code that is used to log the program execution.

After compiling GRaCCE with the profiling flag, it is run with only the FS option chosen. *Gprof* is only used on the FS routine using the Iris data set due to the considerable overhead profiling adds.

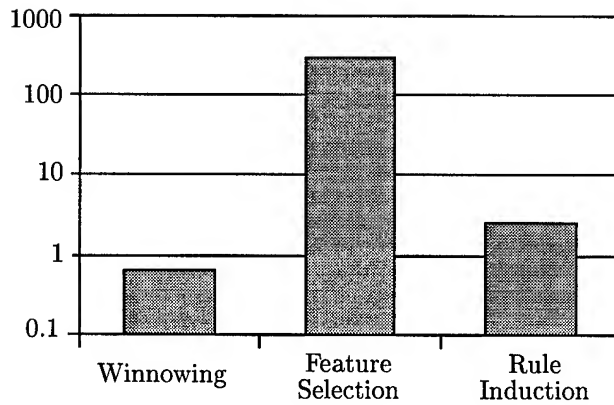


Figure 8: Serial GRaCCE Timing

What normally took a couple of minutes to execute, required several hours with the added profiling overhead. With the profiling added, execution of GRaCCE creates a log file called *gmon.out*. The log file contains information pertaining to only those sections of GRaCCE that actually get executed during the run. Because only FS is chosen, no output is generated for the winnowing or rule induction phases.

Once the log file is made, the actual gprof program is called to interpret the results. Gprof is invoked with the following command line:

```
gprof OPTIONS executable logfile
```

Gprof prints a flat profile and a call graph. A short section of the GRaCCE flat profile with just the important lines extracted pertaining to is shown in Figure 9 with a much larger section of the output in the Appendixes. Appendix C contains the flat profile output and Appendix D contains the call graph output. One should note the large percentage of time that the program is executing in the TNT library. A large performance improvement could be made with a more efficient matrix library.

The FS structure of GRaCCE can be seen in Figure 9 by moving from the bottom line back up to the top line. At program start, FS is selected from the menu which calls *gracce_fs.el*. Next,

Each sample counts as 0.01 seconds.						
%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
3.29	64.40	3.86	28500	0.14	4.10	sortbydist2(bool, ...
0.09	117.04	0.11	190	0.58	617.69	knn2(unsigned int, ...
0.00	117.42	0.00	21	0.00	5589.63	objfun2(unsigned int, ...
0.00	117.42	0.00	1	0.00	117417.65	ga_fs01(unsigned int, ...
0.00	117.42	0.00	1	0.00	117417.65	gracce_exop(void)
0.00	117.42	0.00	1	0.00	117417.65	gracce_fsel(void)

Figure 9: Significant Profiler Lines for GRaCCE

gracce_exop is called which in turn calls *gafs01*, the strictly GA method of feature selection. As can be seen, these three calls are made only once during the program execution.

The *ga_fs01* routine, however, makes a total of twenty-one calls to *objfun2*. This number is directly related to the population size of the GA. There is one initial call to the objective (or fitness) function and then a call for every generation. For this testing the population size is set to ten and the GA runs for twenty generations. The total is twenty-one calls to the objective function as can be seen in Figure 9.

Next, for each member of the population that it receives, *objfun2* calls the *knn* function, which is a *k* nearest neighbor algorithm. So how does the total number of calls to kNN come out to 190? This introduces a quirk noticed during analysis but which doesn't affect the results when compared to the original Windows C++ implementation. With a population size of ten the actual execution shows that the initial call to *objfun2* passes a population of size ten and then each subsequent generation passes a population of only nine members. This explains how the total number of calls to kNN comes out to 190. Finally, *sortbydist2* is called 150 times for each call to kNN which corresponds to the number of sample vectors in the iris data set.

4.1.2 Serial Timing Baseline

Before going to the next step of implementing a parallel version of GRaCCE, a baseline is needed for comparing with the parallel implementation. Several different data sets were chosen for testing and their characteristics were previously shown in Table 3. The timing results for several of these data sets using the serial version of GRaCCE on PCs with different processor speeds is shown in

Table 4.

	333 MHz			400 MHz		
	Mean (s)	Median (s)	StdDev (s)	Mean (s)	Median (s)	StdDev (s)
Iris	217	217	3.57	182	182	3.16
Wine	330	332	9.50	275	274	10.7
Glass	469	472	12.1	392	391	7.03
Cancer	4663	4645	103	3888	2883	67

	600 MHz			933 MHz		
	Mean (s)	Median (s)	StdDev (s)	Mean (s)	Median (s)	StdDev (s)
Iris	121	121	1.48	77.5	77.5	1.04
Wine	185	187	4.47	117	117	3.64
Glass	264	266	7.12	166	166	2.82
Cancer	2535	2520	53	1723	1717	31.0

	1000 MHz		
	Mean (s)	Median (s)	StdDev (s)
Iris	72.4	72.9	1.05
Wine	111	111	2.64
Glass	157	159	3.37
Cancer	1615	1611	27.5

Table 4: Serial Execution Baselines by Processor Speed

4.2 Parallel Implementation

Once the decision is made to parallelize the objective function within the feature selection GA, the serial code was modified using the MPICH library functions. The GA internal operation is shown in Figure 10. The first design decision is to minimize the amount of code nodes other than the root node are required to run. With MPI, every node executes the same code starting from the initialization point. Since only a small section of the feature selection GA is executed in parallel, this is the only necessary code for each non-root node. Minimizing the amount of processing each node does before reaching the objective function is accomplished by placing an *if* statement around the original code in the `gracce.cpp` file. If the process is root, then it executes all of the GRaCCE code. If not root, then each node starts executing only after the root process calls the objective function.

Next, the common data required by all nodes is sent out by the root node to all other nodes. The method used to accomplish this is the `MPI :: COMM_WORLD.Bcast()` method which is a one to

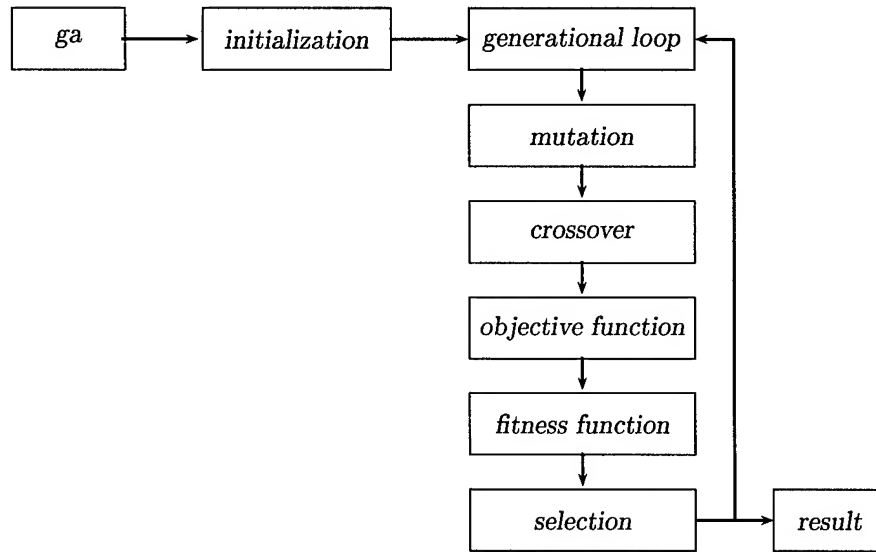


Figure 10: Feature Selection Genetic Algorithm

all broadcast and is more efficient than the root node sending all the data to each node individually [25]. The main broadcast at this point is the complete sample data set that gets sent to all nodes when the data set is loaded in from its file. A few other global system variables are also broadcast at this time. Accomplishing this at start-up reduces the communications burden in the following iterative sections of the GA.

At this point it is necessary to decide upon the approach used to parallelize the objective function internals. Two approaches are considered:

1. Parallelize the k-nearest-neighbor (kNN) algorithm used to evaluate each population member,
and
2. Spread the population members out to the nodes for concurrent processing.

The selected approach is taken based on the iterative nature of the objective function. The objective function calls the kNN algorithm for each member in the population. The first approach results in iteratively calling a parallel kNN for each member. The second approach results in a parallel distribution of the population to individual nodes that run the kNN algorithm sequentially. The difference is shown in Figure 11.

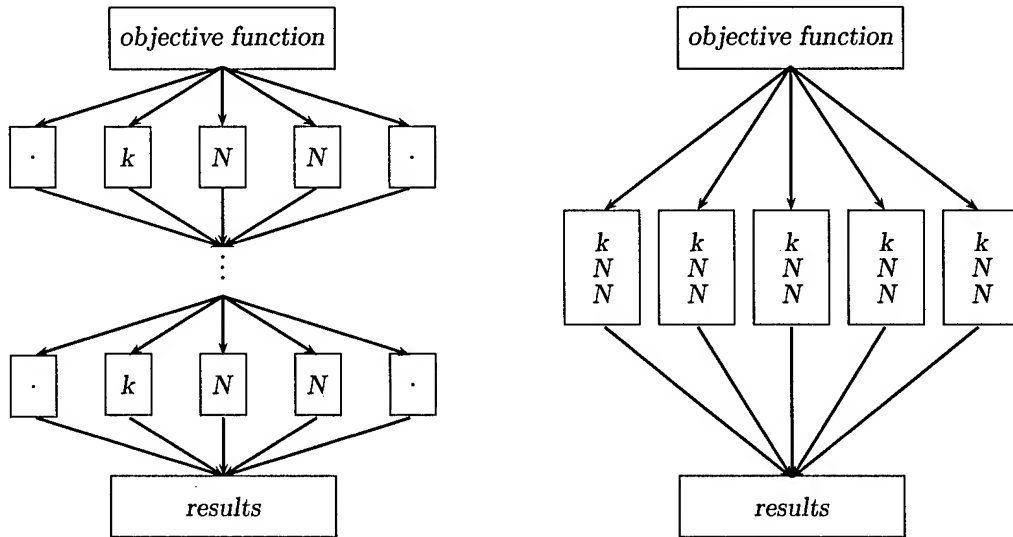


Figure 11: Possible Parallel Objective Function Approaches

The first method is much more complicated than the second method in that the kNN algorithm has to be rewritten. Additionally, the first method requires more communications and thus has a greater parallelization overhead. The second method does not require recoding of the kNN algorithm and in fact requires very little modification of the GRaCCE code. Based on this, the second method is chosen.

Using the second method, when the objective function is called the root node first sends out all common data to the other nodes by calling the `MPI :: COMM_WORLD.Bcast()` function for all necessary items. This amounts to a couple of integer values and a couple of vectors containing the population members. Once the data is received, each node's processing is completely independent of the other nodes. Each node then executes the objective function concurrently on the population members it is responsible for without any further communication between nodes. When each node is finished it sends the results it calculated back to the root node which puts all of the results together and returns from the objective function call.

One important consideration regarding the AFIT cluster is the heterogeneous nodes. This can cause problems when using MPI as the communications mechanism. The MPI function calls time out and exit with errors if they have to wait too long for the matching call in another node. Normally, on

a homogeneous system, an implementation such as this one is not a problem due to all of the nodes will finish the kNN algorithm at nearly the same time. However, in a heterogeneous environment, a processor that is twice as fast as another node finishes its work in roughly half the time. If this difference is large enough the MPI calls will time out.

In order to alleviate this problem, all nodes are required to wait at a `MPI :: COMM_WORLD.Barrier()` call before sending their results back to the root node. This allows different speed processors to be used without the `MPI :: COMM_WORLD.Send()` calls timing out and generating error messages if not actually crashing the execution.

Now that the objective function is implemented, it's time to test how efficient it is. With a serial baseline for comparison, the parallel version of GRaCCE's timing results are gathered and analysed. The root processor for the parallel testing on up to twenty nodes is chosen to be one of the 333 MHz computers. With the MPI Barriers in the GRaCCE code this provides the appearance of a cluster of homogeneous 333 MHz processors.

The results of testing the parallel implementation of GRaCCE using the Iris data set are shown in Figure 12. The results are shown for five, ten, fifteen, and twenty processors along with the original timing of the serial version of GRaCCE. As expected the time required for feature selection decreases as processors are added. However, it is noted that there is an exception to this between ten and fifteen processors. The difference between the mean execution time of the ten and fifteen processors is statistically insignificant. This is explained later in this section.

The results for the tests using the Wine, Glass, and Cancer data sets are shown in Figures 13, 14, and 15 respectively. All of the figures have similar results, with only the length of time scaled between them. The corresponding plots of speedup and efficiency for the data sets tested are shown in Figures 16 and 17. The isoefficiency is not calculated for reasons stated in [14], the primary one essentially concluding that GRaCCE's stochastic nature tends to make the isoefficiency calculations meaningless. For instance, two data sets of the same size may have significantly different efficiencies due to the stochastic nature of the search space.

Fifteen processors showed no gain in speedup or efficiency over ten processors because of the

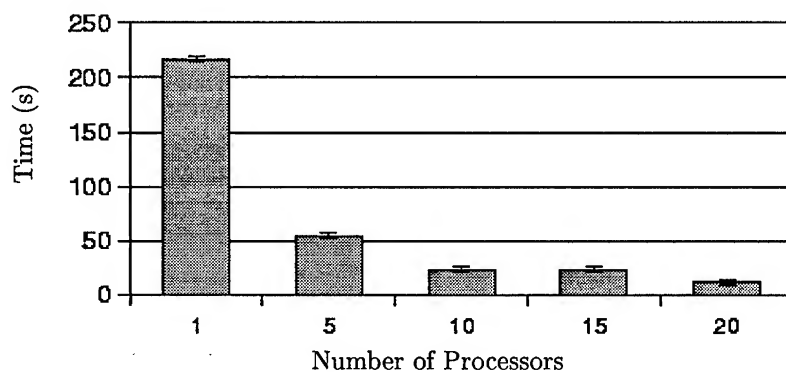


Figure 12: Iris Data Set Test Results

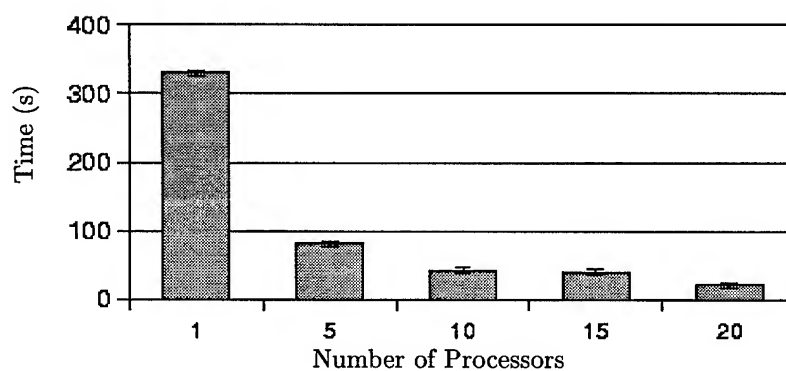


Figure 13: Wine Data Set Test Results

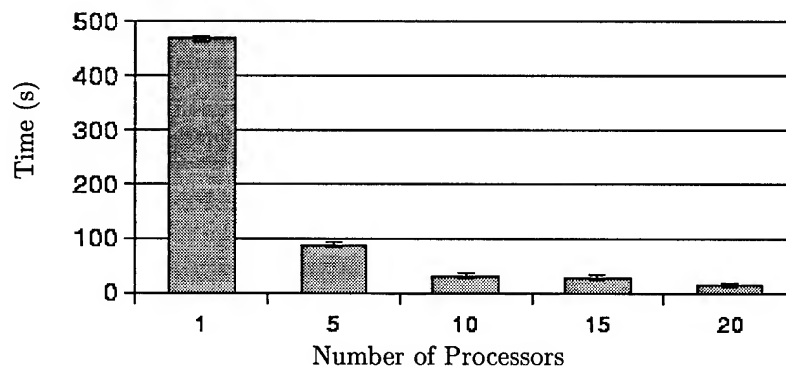


Figure 14: Glass Data Set Test Results

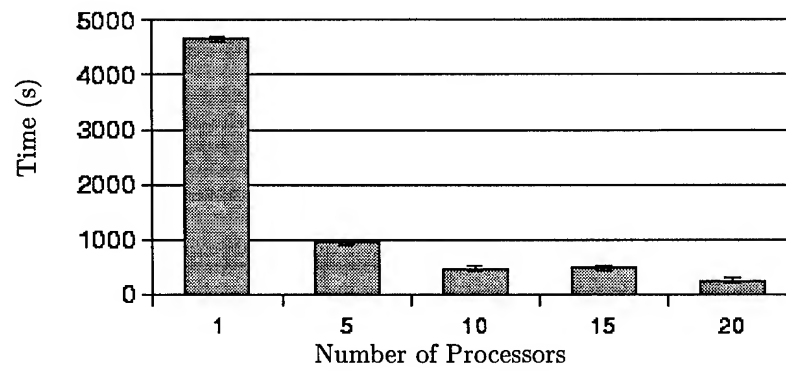


Figure 15: Cancer Data Set Test Results

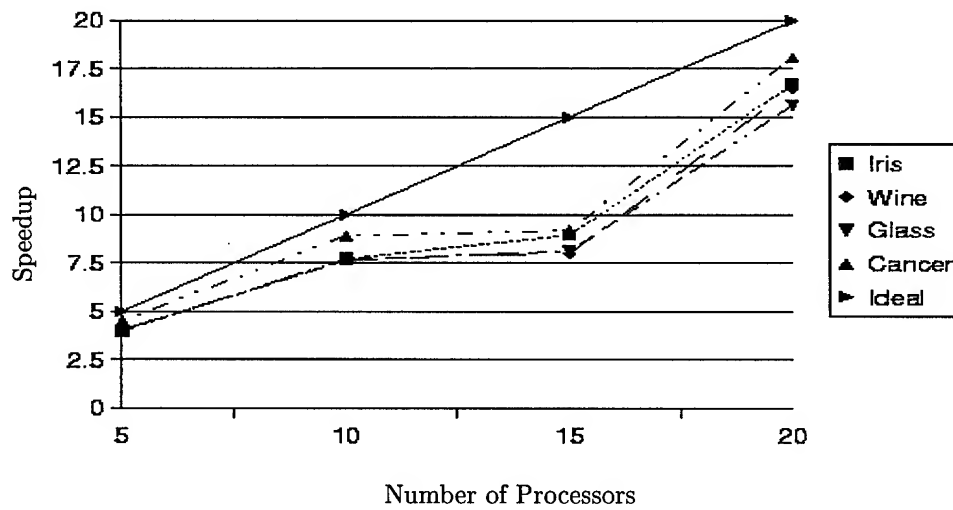


Figure 16: Speedup

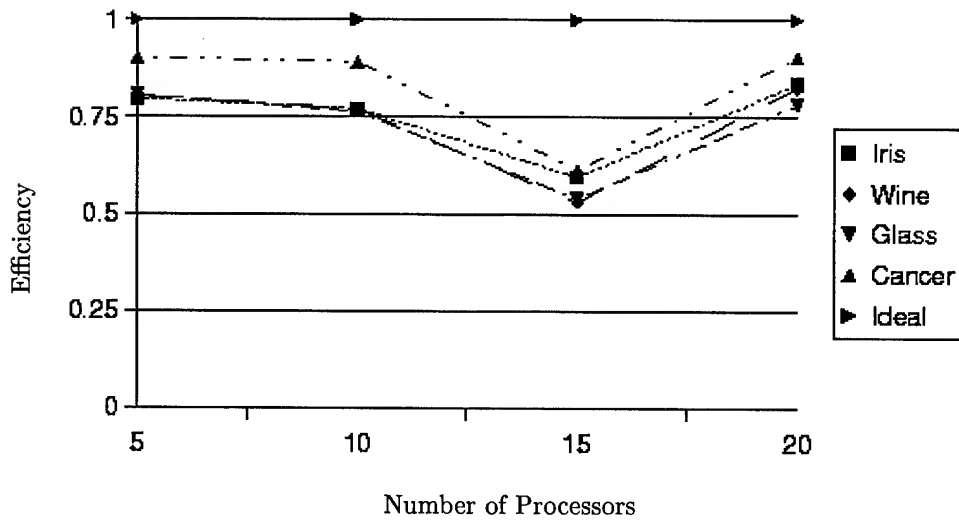


Figure 17: Efficiency

algorithm used to distribute the workload and the treatment of the AFIT cluster as a set of homogeneous systems. The results are based on a population size of twenty. Thus if two processors are used each one gets ten members to evaluate. With five processors, each one gets four members to evaluate and with ten processors, each node gets two members to evaluate.

However, with fifteen processors the workload cannot be evenly divided among the nodes. Some of the nodes get two members and some get only one member. By treating the whole cluster as a set of similar nodes the end result is the total time is set by the nodes that receive two members to evaluate. This results in the same statistical performance achieved by the ten processor case.

This is further demonstrated by Figure 18. This set of data is collected on a set of five homogeneous 1GHz computers. The population size of the GA is varied from one to twenty members and the number of processors is varied from two to five. A clear stair-stepping is seen in the results. The stair-stepping is not perfect due to the algorithm MPICH uses to allocated the processes to processors. However, the results are consistent and the stair-stepping pattern is the same for each of the ten runs made during testing.

With the results of these tests on the parallel implementation of the objective function, the population size should be made an integer multiple of the number of processors used. At integer

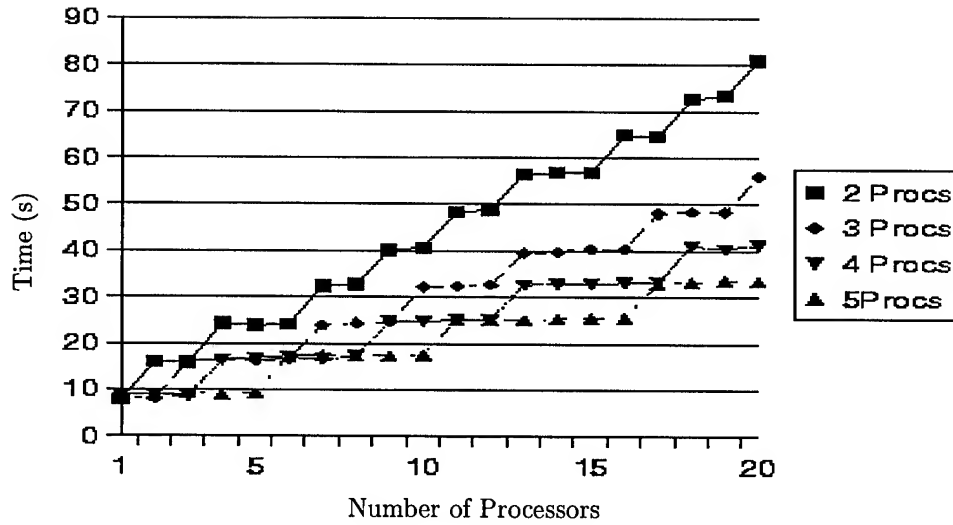


Figure 18: Iris with Multiple Populations

multiples, all of the processors are fully utilized. Anything other than an integer multiple results in idle processors and less efficient performance.

4.3 Heterogeneous Environments

An additional modification to the parallel implementation is made to allow a better utilization of the processing nodes, taking into account the heterogeneous nature of the AFIT cluster. First, each parallel process is enabled to open the `/proc/cpuinfo` file and then to find and read in the *bogomips* rating for the node it is executing on. Each node reports this value to the root processor. The root processor sorts the nodes by this value and assigns work starting with the fastest node first. The algorithm starts at one and increments until the population size is reached. At each increment it adds one to the node it is at and then moves to the next one in a ring type manner.

Even with such a simple fastest-first algorithm, a clear improvement, see Figure 19, in execution time is gained over treating the nodes as homogeneous. This holds except for certain population sizes, where the slowest processors are required. The test setup uses two 333MHz computers, two 450MHz computers and a 933MHz computer. All of the test runs are started with the root node on one of the 333MHz computers. This constrains the homogeneous code to always assigning work to

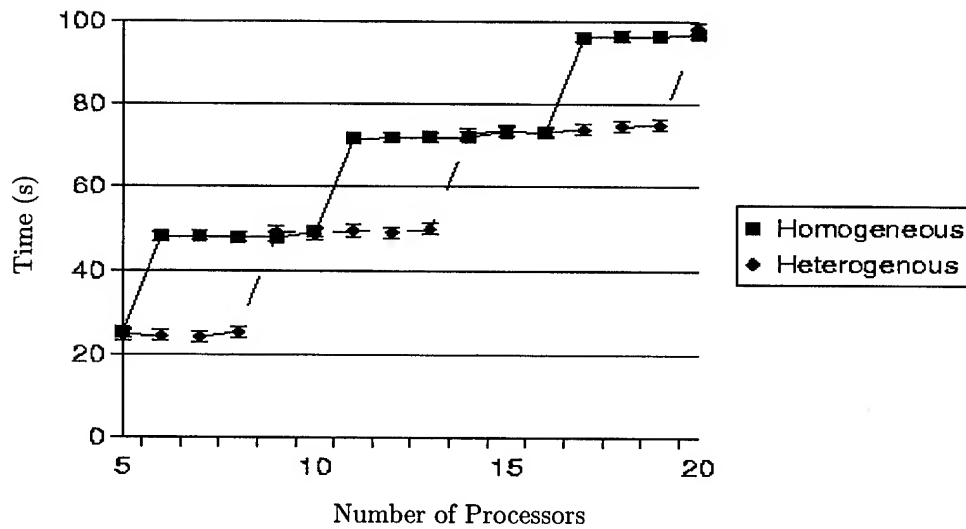


Figure 19: Effects of Heterogeneous Algorithm

the slowest computers first. With the heterogeneous sorting of the nodes, the root node will assign work to the fastest nodes first, even if that results in the root node remaining idle.

4.4 Summary

In this chapter, the parallel improvement to GRaCCE is developed with the resulting implementation tested and analyzed. First is the analysis of the serial GRaCCE code targeting any performance bottlenecks in the code. Parallel improvements are explored and the decision to implement a parallel objective function is made. Implementation issues are then discussed followed by the test and analysis of the implementation that is chosen. The final conclusions drawn from the developed code and the experiments on it are made in the following chapter.

5 Conclusions and Recommendations

This chapter presents a summary of this thesis effort. The significant contributions of this effort are presented and recommendations on future research are provided.

5.1 Summary

Following a brief introduction, Chapter 2 provides a more detailed explanation of data mining and its importance to extracting “the golden nuggets” of information from today’s ever growing databases. This is followed by an introduction and description of the genetic rule and classifier construction environment (GRaCCE). Chapter 2 concludes with a discussion of important parallel computing issues from design considerations to performance metrics.

Chapter 3 presents design and implementation issues relating to this thesis effort. First is the consideration of the hardware, software and operating system to use for this effort and the impacts of those decisions. The issues for analyzing the serial version of GRaCCE are presented next, followed by parallel design, implementation and testing issues. The chapter concludes with a discussion of differences between heterogeneous and homogeneous architectures and environmental control considerations.

Chapter 4 details the results and analysis of design and implementation decisions made throughout this effort. The serial analysis points to a significant bottleneck in GRaCCE’s feature selection (FS) routines. This is further narrowed down to the objective function within the genetic algorithm. Testing of the serial version also provides a baseline for comparison against any changes in the code. Implementation issues are discussed, followed by experimental test and analysis of the implementation decided upon. The conclusions made regarding the results of this effort are in the following section.

5.2 Conclusions

This thesis effort shows that modification of GRaCCE, through the implementation of part of its code in parallel, results in a clear improvement in efficiency and speed of execution. The section of code chosen is Feature Selection due to its disproportionate execution time relative to other functions

of GRaCCE. The results clearly show that the parallel implementation of GRaCCE achieves the highest speedup and efficiency when the population size, of the genetic algorithm (GA) used, is an integer multiple of the number of nodes. At integer multiples, the speedup achieved is within twenty-five percent of perfect linear speed-up.

Additional performance is gained by taking into consideration the heterogeneous nodes in the cluster. Treating the computational nodes as homogeneous results in all nodes constrained to the poorest performer in the group. Ideally, to get better performance, faster nodes should get a greater workload. As a simple test case, the load distribution algorithm in the feature selection GA is modified. The original algorithm divies out the population members like cards in a poker game, until all members are assigned. The arrangement of the nodes is based on the order in which the nodes are initialized by MPI. The new heterogeneous algorithm sorts the nodes by bogomips value, and then divies the population out by assigning members to the fastest nodes first. This simple change in workload assignment results in a clear improvement, in both speedup and efficiency, over the homogeneous algorithm.

5.3 Recommendations

Based on the results of this thesis effort, the following are recommendations for additional research using the parallel implementation of GRaCCE.

1. With faster feature selection now available, data sets with a higher number of features can be tested.
2. Comparative analysis of feature selection using the genetic algorithm method and the other two feature selection methods of GRaCCE. The analysis could include the quality of solution versus execution time to determine relative rates of convergance to a solution.
3. Analysis of the genetic algorithm's performance while varying its different parameters such as mutation, crossover, and selection rates.

4. Develop a more complex algorithm for distribution of the workload in a heterogeneous environment. Distribution could be scaled according to relative differences in processor speeds rather than just the fastest first.

A AFIT Cluster of PCs

Hostname	processor	Physical RAM (MB)	Network speed (Mb/s)	OS 1	OS 2	ip addr	np
Win2000	dual Pentium III Xeon 550	784	1000		W2k	172.8.0.1	2
Linstar	dual Pentium III 600	512	1000	RH6.2		172.8.0.2	2
abc-a3	Pentium III 600	384	1000	RH6.1	W2k	172.8.0.3	1
abc-a4	Pentium III 600	384	1000	RH6.1	W2k	172.8.0.4	1
abc-a5	Pentium III 600	384	1000	RH6.2	W2k	172.8.0.5	1
abc-b5	Pentium III 600	384	100	RH6.1	W2k	172.8.1.5	1
abc-b6	Pentium III 600	384	100	RH6.1	W2k	172.8.1.6	1
abc-b7	Pentium III 600	384	100	RH6.1	W2k	172.8.1.7	1
abc-b8	Pentium III 600	384	100	RH6.1	W2k	172.8.1.8	1
abc-b9	Pentium III 600	384	100	RH6.1	W2k	172.8.1.9	1
abc-b10	Pentium II 400	256	100	RH6.1		172.8.1.10	1
abc-b11	Pentium II 400	256	100	RH6.1		172.8.1.11	1
abc-b12	Pentium II 400	256	100	RH6.1		172.8.1.12	1
abc-b13	Pentium II 333	256	100	RH6.1		172.8.1.13	1
abc-b14	Pentium II 400	128	100	RH6.1		172.8.1.14	1
abc-b15	Pentium II 450	256	100	RH6.1		172.8.1.15	1
abc-b16	Pentium II 400	128	100	RH6.1		172.8.1.16	1
abc-b17	Pentium II 400	128	100	RH6.1		172.8.1.17	1
abc-b18	Pentium II 333	128	100	RH6.2		172.8.1.18	1
abc-b19	Pentium II 333	128	100	RH6.2		172.8.1.19	1
abc-b20	Pentium II 333	256	100	RH6.2		172.8.1.20	1
abc-b21	Pentium II 400	256	100	RH6.2	W2k	172.8.0.21	1
abc-b22	Pentium III 933	256	100	RH6.2	W2k	172.8.0.22	1
abc-b23	Pentium III 933	256	100	RH6.2		172.8.0.23	1
abc-b24	Pentium III 933	256	100	RH6.2		172.8.0.24	1
abc-b25	Pentium III 933	256	100	RH6.2		172.8.0.25	1
abc-b26	Pentium III 1000GHZ	256	100	RH6.1		172.8.0.26	1
abc-b27	Pentium III 1000GHZ	256	100	RH6.1		172.8.0.27	1
abc-b28	dual Pentium III xeon 550	512	100	RH6.1		172.8.0.28	2
						Total	30

Table 5: AFIT's Pile of PCs

B Evolutionary Algorithms

A general framework for evolutionary algorithms was developed by Thomas Bäck [2]. This framework is based on several aspects of evolutionary algorithms. The first is that evolutionary algorithms utilize a population of individuals. Children are created from these individuals by a randomization process intended to model genetic mutation and/or recombination. From generation to generation these individuals are evaluated based on some quality or fitness function that can be applied to each one of them.

As a general framework for these basic instances, Bäck defines I to denote an arbitrary space of individuals $\alpha \in I$, and $F : I \rightarrow \mathbb{R}$ to denote a real-valued fitness function of individuals. Using μ and λ to denote parent and children population sizes, $P(t) = (\alpha_1(t), \dots, \alpha_\mu(t)) \in I^\mu$ characterizes a population at generation t . Selection, mutation, and recombination are operators that transform complete populations. Bäck describes these operators from a high-level perspective of the overall population but they can be used at the lower level of operating on the individual members of the population. Bäck reduces his representation to a simple recombination-mutation-selection loop as shown in figure 20.

Input: $\mu, \lambda, \Theta_i, \Theta_r, \Theta_m, \Theta_s$
Output: a^* , the best individual found during the run or P^* , the best population found during the run.

```

1   $t \leftarrow 0$ ;
2   $P(t) \leftarrow \text{initialize}(\mu)$ ;
3   $F(t) \leftarrow \text{evaluate}(P(t), \mu)$ ;
4  while ( $\iota(P(t), \Theta_i) \neq \text{true}$ ) do
5       $P'(t) \leftarrow \text{recombine}(P(t), \Theta_r)$ ;
6       $P''(t) \leftarrow \text{mutate}(P'(t), \Theta_m)$ ;
7       $F(t) \leftarrow \text{evaluate}(P''(t), \lambda)$ ;
8       $P(t+1) \leftarrow \text{select}(P''(t), F(t), \mu, \Theta_s)$ ;
9       $t \leftarrow t + 1$ ;
10 od
```

Figure 20: Bäck's Basic Evolutionary Algorithm

The initial population is instantiated by some chosen means which may be deterministic or stochastic. The initial population is then evaluated based on a chosen fitness or quality measure.

Once this is done, the while loop is entered. The methods of recombination, mutation, evaluation, and selection are the distinguishing characteristics between the major categories of evolutionary algorithms and are discussed in the next sections.

The while loop is repeated until the termination condition is met. The termination condition varies from implementation to implementation and some possibilities are a certain number of generations, a certain quality of the overall population, or the population has converged.

B.1 Genetic Algorithms

Genetic algorithms are a class of evolutionary algorithms first proposed and analyzed by John Holland[2]. There are three features which distinguish GAs as first proposed by Holland from other evolutionary algorithms:

1. The representation used, which is a bit string,
2. the method of selection, which is proportional selection, and
3. the primary method of producing variations, which is crossover.

Of these three features, however, it is the emphasis placed on crossover which makes GAs distinctive. Since it's original proposal, other alternative methods of representations and methods of selection have been used.

Although many methods of crossover have been proposed, in almost every case these variants are made in the spirit of Holland's original GA behavior in the processing of schemata into building blocks. Individual structures are referred to as chromosomes. They are the genotypes that are manipulated by the genetic algorithms. Evaluation routine decodes these structures and assigns a fitness value. Typically, but not necessarily, the chromosomes are bit strings. The value at each locus is commonly referred to as an allele. Sometimes the individual loci are called genes. At other times the genes are combinations of alleles that have some phenotypical meaning.

Based on Bäck's general evolutionary algorithm outline the first thing is to create an initial population. This initial creation can be random or deterministic. Once they are created an initial

fitness evaluation is made and then the main loop begins to iterate until some termination criteria is met.

There are several methods for selecting which members of the population are used to generate the children in the next generation. In the original GA that Holland proposed, individuals are chosen for mating probabilistically. This biases the reproduction in favor of individuals with a better fitness. This is similar to nature's survival of the fittest. The original GA had only one pair selected for each mating cycle. Once the individuals that are chosen for mating of have been selected the genetic operators are then applied. These operators are mutation and crossover and can be applied at application specific rates.

In addition to selecting which members are the parents, a determination has to be made as to how many children will get created from that mating. Some GAs produce one child per set of parents while others can produce many children from one parent. The algorithm is considered to be a steady state GA when only a few children are generated and the majority of the population remains constant or stays the same.

Once the children have been generated, the selection operator is used to produce the next generation. One should take care in how the selection is biased. Incorrect biasing may lead to pre-convergence of the search and entrapment in a local minimum or maximum.

B.2 Representations

For every problem there has to be some way to represent the problem domain in a way that can be understood by a computer. This representation [2] is very important regarding how the algorithm is constructed and how it performs. As the structure of this representation varies from problem to problem so may the actual representation vary in a number of ways. The choice of representation depends not only on the problem being represented but also on how that representation is processed. For instance, the particular search algorithm used may be more efficient on a particular data structure such as parse trees vs. real-valued vectors. Several representations used in evolutionary computing are discussed in the following sections.

Binary Strings

The representation used in Goldberg's original GA is a binary string [2]. These binary strings are typically of fixed length. The mutation operator typically inverts individual bits within the string. The crossover operator has many variations but generally involves swapping matching bit sections between two individuals. Binary strings may use standard binary decoding or may use a Gray code interpretation of the string. Binary strings may also be mapped into some schema used to develop or preserve good building blocks within the strings.

Real-valued Vectors

If the problem domain is based on real-valued functions then a representation that uses real numbers instead of bits is more appropriate. Evolution strategies and evolutionary programming typically use real-valued vectors [2]. When using real values the concept of building blocks is not used, instead the individual is viewed in its entirety. The typical mutation operator on real-valued individual is to add a multivariate zero-mean Gaussian random variable to it to produce a new child. Real valued vectors can be easily expanded to include some of the evolutionary computation parameters to allow a means of self-adaptation.

Parse Trees

Parse trees [2] are a good fit when the representation desired is for a program or a function. This is a representation that helps to ensure that only syntactically correct programs are created. This greatly reduces the overall size of the search space vs. an unconstrained representation. One advantage of parse trees is its natural recursive definition, which allows the structure to be dynamically changed. One drawback to parse trees, however, is their inability to represent iterative loops.

Other Representations

Other representations[2] include the following:

1. Mixed-Integer Representation,

2. Introns, and

3. Diploid Representation.

A mixed-integer representation allows a combination of binary strings and real-valued vectors within an individual member of the population. Introns use additional encoding within the genotypical representation based on the biological concept of introns and exons in the chromosomes. This results in sections of the representation that may or may not contribute to the resulting decoding of the individual solution. Finally, the diploid representation allows multiple allele values for each position in the genome. An additional mechanism is required to specify which one is dominant.

B.3 Selection Operators

Selection [2] is one of the main operators used in evolutionary computing and its primary function is to evolve the population towards a better solution. There are two places in the general EC algorithm where selection occurs. The first is the selection of which members get selected to produce children. The second is which members in the resulting population get to survive in the next generation. The following sections discuss some of the main selection schemes used.

Proportional Selection

A proportional selection [2], or roulette wheel selection, makes a selection based on a proportionally assigned weight. The proportion for a poor individual is small and the proportion assigned to the best individual is the largest. All other individuals have proportions in between. This selection may be made stochastically or deterministically. For a stochastic selection a roulette wheel can be used to probabilistically select an individual, with each place on the wheel having a probability sized proportionally to the fitness of the individual. For a deterministic selection, each individual is selected for reproduction a proportional number of times.

Tournament Selection

A tournament [2] is used to select one out of a group of competing individuals. The size of the group has to be greater than one for a competition to take place. The individuals selected for a tournament may be chosen deterministically or stochastically. This is closer to a survival of the fittest selection.

Rank-based Selection

The rank-based selection [2], also known as elitist selection, ranks the population by some criteria. The criteria is commonly the individuals fitness value. This selection starts with the best individual and moves down the list until the desired number is chosen. The actual ranking may be a linear ranking in which the rank of an individual is directly related to its fitness. The ranking might also be a non-linear ranking in which better individuals get a higher weight. A third ranking might have a threshold value in which only individuals with a fitness greater than a set value get selected. This threshold value might be a set value or an adaptive one based on another population value such as the mean population fitness.

B.4 Search Operators

Search operators [2] make changes to individuals in the population in a manner that moves the population members around within the search space. These operators have to be aware of the representation used within the individual members in order to maintain the functionality of the representation. Regardless of whether mutation or some form of recombination is used, the resulting individual still has to be interpreted within the space being searched and as it relates to the problem domain. Mutation leans more to the exploration side of evolutionary computation and recombination leans more to the exploitation side of evolutionary computation.

Mutation

Mutation [30] can be thought of as nature's way of trying something new. Point mutations and regulatory mutations are two types of mutation that occur in nature. A point mutation occurs when

a base gets added into a DNA strand which results in possible changes in the codon sequences. Regulatory mutations result in changes as to whether or not a particular gene sequence is active. Sometimes the new mutation survives, sometimes it gets repaired, and sometimes it just dies out.

Generally, for every representation used in evolutionary computations there is a mutation operator [2] that has been designed to operate on it. For instance, with binary strings a mutation would result in a bit flip at some location in the bit string. For a real-valued vector, a mutation might be the addition or subtraction of a normally distributed random variable within some bound. For a permutation representation a mutation operator, such as the 2-opt, might pick two points in the sequence and reverse all the points in between.

A possibly new mutation operator is essentially a lower level simulation of what happens in nature when a point mutation occurs. As described previously, an additional base is inserted into the existing DNA strand. This results in a shifting of the codons used in generating the proteins from the messenger RNA. This can be simulated by applying a random mutation to a string. If the mutation occurs then a random bit gets inserted at that point and the rest of the bits in the string get shifted. To keep the size of the binary strings constant, the final bit gets thrown away.

Recombination

Recombination [2] can be thought of as mixing the genetic pot by swapping alleles between two chromosomes. Two (or more) individuals are combined in some manner as to create new children with only the parts that existed in the parents. The number of parents used and the number of children created varies depending upon the designers choice.

Like mutation, just about every representation has some form of recombination operator designed for it. A binary string crossover can number from one to many crossover points. Instead of having predetermined crossover points the crossover points can be determined dynamically. A uniform crossover gives every bit location in the string an equal weight and then iteratively determines whether each bit location gets to be a crossover point. The other representations have similar operations tailored to their underlying representation.

C GProf Flat Profile of GRaCCE FS

The following is the flat profile output from the GNU profiler *gprof* for a single run of GRaCCE. The feature selection option was chosen and executed on the Iris data set. Lines with a *total ms/call* greater than four seconds are used in determining the most significant functions. However, particular attention should be paid to the high percentage of time the program executes within the Template Numerical Toolkit's Matrix initialization, copy, and destroy functions.

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
 listing.

calls the number of times this function was invoked, if
 this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
 else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
 function is profiled, else blank.

name the name of the function. This is the minor sort
 for this listing. The index shows the location of
 the function in the gprof listing. If the index is
 in parenthesis it shows where it would appear in
 the gprof listing if it were to be printed.

Each sample counts as 0.01 seconds.

%	cumulative	self	self	total		
time	seconds	seconds	calls	ms/call	ms/call	name
24.60	28.89	28.89	55449967	0.00	0.00	TNT::Matrix<double>::initialize(i...
11.24	42.09	13.20	46945595	0.00	0.00	TNT::Matrix<double>::copy(double ...
7.66	51.09	9.00	64030514	0.00	0.00	TNT::Matrix<double>::destroy(void...
4.33	56.17	5.08	4246500	0.00	0.00	TNT::Matrix<double> TNT::matmult<...
3.72	60.54	4.37	17131578	0.00	0.00	TNT::Matrix<double>::set(double c...
3.29	64.40	3.86	28500	0.14	4.10	sortBydist2(bool, double, TNT::Ma...
3.17	68.12	3.72	44119119	0.00	0.00	TNT::Matrix<double>::operator()(i...
3.04	71.69	3.57	4246500	0.00	0.00	TNT::Matrix<double> operator-<dou...
2.79	74.97	3.28	25567272	0.00	0.00	TNT::Matrix<double>::operator=(TN...
2.78	78.24	3.27	45496310	0.00	0.00	TNT::Matrix<double>::operator()(i...
2.71	81.42	3.18	4246521	0.00	0.00	TNT::Matrix<double> TNT::transpos...
2.21	84.02	2.60	47090598	0.00	0.00	TNT::Matrix<double>::~~Matrix(void...
2.17	86.57	2.55	52919572	0.00	0.00	TNT::Index1D::lbound(void) const
1.78	88.66	2.09	4333608	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...

1.75	90.71	2.05	8522216	0.00	0.00	TNT::const_Region2D<TNT::Matrix<d...
1.64	92.63	1.92	21378323	0.00	0.00	TNT::Matrix<double>::Matrix(TNT::...
1.59	94.50	1.87	35270610	0.00	0.00	TNT::Matrix<double>::operator[] (i...
1.43	96.18	1.68	17131578	0.00	0.00	TNT::Matrix<double>::Matrix(int, ...
1.41	97.84	1.66	23698500	0.00	0.00	TNT::const_Region2D<TNT::Matrix<d...
1.41	99.49	1.65	4246500	0.00	0.00	TNT::Matrix<double> map_sqrt<do...
1.21	100.91	1.42	28500	0.05	0.12	TNT::Matrix<double> sortNSmallest...
1.18	102.29	1.38	17133090	0.00	0.00	TNT::Matrix<double>::num_rows(voi...
1.06	103.53	1.24	16994215	0.00	0.00	TNT::Matrix<double>::operator() (i...
0.96	104.66	1.13	26459826	0.00	0.00	TNT::Index1D::ubound(void) const
0.96	105.79	1.13	16020167	0.00	0.00	TNT::Matrix<double>::operator[] (i...
0.95	106.91	1.12	21378356	0.00	0.00	TNT::Matrix<double>::num_cols(voi...
0.95	108.02	1.11	13725860	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.90	109.08	1.06	4704976	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.89	110.12	1.04	13344733	0.00	0.00	TNT::Index1D::Index1D(int, int)
0.76	111.01	0.89	4247176	0.00	0.00	VecQueue<double>::dequeue(void)
0.75	111.89	0.88	28501	0.03	0.88	VecQueue<double>::buildByRows(void)
0.73	112.75	0.86	4247176	0.00	0.00	VecQueue<double>::enqueue(TNT::Ma...
0.44	113.27	0.52	8522216	0.00	0.00	TNT::Matrix<double>::operator() (T...
0.37	113.70	0.43	8580547	0.00	0.00	TNT::Matrix<double>::Matrix(void)
0.37	114.13	0.43	8522216	0.00	0.00	TNT::const_Region2D<TNT::Matrix<d...
0.35	114.54	0.41	8522216	0.00	0.00	TNT::const_Region2D<TNT::Matrix<d...
0.32	114.92	0.38	4247176	0.00	0.00	VecNode<double>::~VecNode(void)
0.31	115.28	0.36	4247176	0.00	0.00	VecNode<double>::~VecNode(TNT::Mat...
0.26	115.58	0.30	4704976	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.26	115.88	0.30	4305073	0.00	0.00	TNT::Index1D::operator=(TNT::Inde...
0.24	116.16	0.28	13914909	0.00	0.00	TNT::Matrix<double>::lbound(void)
0.23	116.43	0.27	4246500	0.00	0.00	TNT::Matrix<double> TNT::operator...
0.16	116.62	0.19	4275867	0.00	0.00	VecQueue<double>::isEmpty(void) c...
0.14	116.78	0.16	4704976	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.13	116.93	0.15	4704976	0.00	0.00	TNT::Matrix<double>::operator() (T...
0.09	117.04	0.11	190	0.58	617.69	knn2(unsigned int, TNT::Matrix<un...
0.08	117.13	0.09	171000	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.05	117.19	0.06	29026	0.00	0.01	TNT::Matrix<double> mat<double>(T...
0.03	117.23	0.04	771710	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.03	117.27	0.04	29125	0.00	0.01	TNT::Matrix<double> mat<double>(T...
0.02	117.29	0.02	28960	0.00	0.00	TNT::Matrix<int>::set(int const &)
0.02	117.31	0.02	28691	0.00	0.00	VecQueue<double>::~VecQueue(void)
0.02	117.33	0.02	28691	0.00	0.00	VecQueue<double>::VecQueue(void)
0.02	117.35	0.02	28500	0.00	0.00	int maxPos<int>(TNT::Matrix<int> ...
0.01	117.36	0.01	265047	0.00	0.00	TNT::Matrix<int>::operator() (int)
0.01	117.37	0.01	171159	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...
0.01	117.38	0.01	28837	0.00	0.00	TNT::Matrix<double>::size(void) c...
0.01	117.39	0.01	12852	0.00	0.00	TNT::Matrix<unsigned int>::operat...
0.01	117.40	0.01	3533	0.00	0.00	TNT::Matrix<unsigned int>::initia...
0.01	117.41	0.01	620	0.02	0.02	TNT::Matrix<int>::initialize(int,...
0.01	117.42	0.01	580	0.02	0.02	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	107093	0.00	0.00	TNT::Matrix<int>::operator() (int)...
0.00	117.42	0.00	28691	0.00	0.00	VecQueue<double>::queueSize(void)...
0.00	117.42	0.00	28691	0.00	0.00	VecQueue<double>::vecSize(void) c...
0.00	117.42	0.00	28520	0.00	0.00	TNT::Matrix<int>::size(void) const
0.00	117.42	0.00	28500	0.00	0.00	TNT::Matrix<int>::operator=(int c...
0.00	117.42	0.00	11528	0.00	0.00	TNT::Matrix<unsigned int>::operat...
0.00	117.42	0.00	8960	0.00	0.00	TNT::Matrix<unsigned int>::operat...
0.00	117.42	0.00	5520	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	4902	0.00	0.00	TNT::Matrix<unsigned int>::lbound...
0.00	117.42	0.00	4480	0.00	0.00	TNT::Matrix<unsigned int>::operat...
0.00	117.42	0.00	4120	0.00	0.00	TNT::Matrix<unsigned int>::operat...
0.00	117.42	0.00	3740	0.00	0.00	TNT::Matrix<unsigned int>::destro...

0.00	117.42	0.00	3680	0.00	0.00	TNT::const_Region2D<TNT::Matrix<u...
0.00	117.42	0.00	3576	0.00	0.00	TNT::Matrix<unsigned int>::~~Matri...
0.00	117.42	0.00	2343	0.00	0.00	TNT::Matrix<unsigned int>::~num_ro...
0.00	117.42	0.00	2323	0.00	0.00	TNT::Matrix<unsigned int>::~num_co...
0.00	117.42	0.00	1987	0.00	0.00	TNT::Matrix<unsigned int>::~copy(u...
0.00	117.42	0.00	1786	0.00	0.00	TNT::Matrix<unsigned int>::~set(un...
0.00	117.42	0.00	1785	0.00	0.00	TNT::Matrix<unsigned int>::~Matrix...
0.00	117.42	0.00	1761	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	1761	0.00	0.00	TNT::Matrix<unsigned int>::~operat...
0.00	117.42	0.00	1761	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	1761	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	1584	0.00	0.00	TNT::Matrix<unsigned int>::~Matrix...
0.00	117.42	0.00	1524	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	1100	0.00	0.00	TNT::Matrix<double>::~operator()(i...
0.00	117.42	0.00	920	0.00	0.00	TNT::const_Region2D<TNT::Matrix<u...
0.00	117.42	0.00	920	0.00	0.00	TNT::Matrix<unsigned int>::~operat...
0.00	117.42	0.00	920	0.00	0.00	TNT::const_Region2D<TNT::Matrix<u...
0.00	117.42	0.00	920	0.00	0.00	TNT::const_Region2D<TNT::Matrix<u...
0.00	117.42	0.00	860	0.00	0.00	TNT::Matrix<unsigned int>::~operat...
0.00	117.42	0.00	760	0.00	0.00	TNT::Matrix<int>::~destroy(void)
0.00	117.42	0.00	670	0.00	0.00	TNT::Matrix<unsigned int>::~size(v...
0.00	117.42	0.00	640	0.00	0.00	TNT::Matrix<int>::~~Matrix(void)
0.00	117.42	0.00	590	0.00	0.00	TNT::Matrix<double> operator/<dou...
0.00	117.42	0.00	540	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	540	0.00	0.00	TNT::const_Region2D<TNT::Matrix<u...
0.00	117.42	0.00	460	0.00	0.02	TNT::Matrix<int>::~Matrix(int, int...
0.00	117.42	0.00	423	0.00	0.00	TNT::Matrix<unsigned int>::~operat...
0.00	117.42	0.00	380	0.00	0.01	TNT::Matrix<unsigned int> mat<uns...
0.00	117.42	0.00	380	0.00	0.00	unsigned int sum<unsigned int>(TN...
0.00	117.42	0.00	360	0.00	0.01	TNT::Matrix<unsigned int> TNT::mu...
0.00	117.42	0.00	260	0.00	0.01	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	260	0.00	0.00	TNT::Region2D<TNT::Matrix<unsigne...
0.00	117.42	0.00	220	0.00	0.01	TNT::Matrix<unsigned int> TNT::op...
0.00	117.42	0.00	207	0.00	0.00	TNT::Matrix<unsigned int>::Matrix...
0.00	117.42	0.00	191	0.00	0.01	TNT::Matrix<double> appendCols<do...
0.00	117.42	0.00	190	0.00	0.05	TNT::Region2D<TNT::Matrix<double>...
0.00	117.42	0.00	190	0.00	0.02	VecQueue<double>::buildByCols(void)
0.00	117.42	0.00	190	0.00	0.10	get_featsubset(int, TNT::Matrix<u...
0.00	117.42	0.00	190	0.00	0.00	TNT::const_Region2D<TNT::Matrix<d...
0.00	117.42	0.00	151	0.00	0.00	num_items(char *)
0.00	117.42	0.00	150	0.00	0.00	TNT::Matrix<double>::Matrix(int, ...
0.00	117.42	0.00	140	0.00	0.00	TNT::Matrix<int>::Matrix(void)
0.00	117.42	0.00	121	0.00	0.01	TNT::Matrix<unsigned int> mat<uns...
0.00	117.42	0.00	121	0.00	0.00	randMatrix(int, int)
0.00	117.42	0.00	120	0.00	0.01	TNT::Matrix<int>::operator=(TNT::...
0.00	117.42	0.00	101	0.00	0.00	TNT::Matrix<double> operator*<dou...
0.00	117.42	0.00	100	0.00	0.00	TNT::Matrix<int>::copy(int const *)
0.00	117.42	0.00	80	0.00	0.00	TNT::operator+(TNT::Index1D const...
0.00	117.42	0.00	80	0.00	0.01	TNT::Matrix<unsigned int> map_com...
0.00	117.42	0.00	80	0.00	0.01	TNT::Matrix<unsigned int> partial...
0.00	117.42	0.00	61	0.00	0.02	TNT::Matrix<unsigned int> cast<do...
0.00	117.42	0.00	61	0.00	0.00	TNT::Matrix<double> map<double, d...
0.00	117.42	0.00	60	0.00	0.01	TNT::Matrix<unsigned int> TNT::op...
0.00	117.42	0.00	60	0.00	0.00	TNT::Matrix<double> cast<unsigned...
0.00	117.42	0.00	60	0.00	0.00	TNT::Matrix<double> TNT::mult_ele...
0.00	117.42	0.00	60	0.00	0.02	TNT::Matrix<int>::newsize(int, int)
0.00	117.42	0.00	60	0.00	0.00	TNT::Matrix<int>::num_rows(void) ...
0.00	117.42	0.00	60	0.00	0.03	void sort<double>(TNT::Matrix<dou...
0.00	117.42	0.00	53	0.00	0.00	TNT::Region2D<TNT::Matrix<double>...

0.00	117.42	0.00	53	0.00	0.00	double sum<double>(TNT::Matrix<do ...
0.00	117.42	0.00	40	0.00	0.02	TNT::Matrix<int>::Matrix(TNT::Mat ...
0.00	117.42	0.00	40	0.00	0.02	TNT::Matrix<unsigned int> map_les ...
0.00	117.42	0.00	40	0.00	0.02	TNT::Matrix<unsigned int> map_les ...
0.00	117.42	0.00	40	0.00	0.02	TNT::Matrix<unsigned int> map_rem ...
0.00	117.42	0.00	40	0.00	0.00	TNT::Matrix<double> rangeVect<dou ...
0.00	117.42	0.00	23	0.00	0.00	TNT::Matrix<double>::newsize(int, ...
0.00	117.42	0.00	21	0.00	0.00	double minVal<double>(TNT::Matrix ...
0.00	117.42	0.00	21	0.00	5589.63	objfun2(unsigned int, TNT::Matrix ...
0.00	117.42	0.00	20	0.00	0.00	TNT::Matrix<double> TNT::operator ...
0.00	117.42	0.00	20	0.00	0.02	TNT::Matrix<unsigned int> map_isE ...
0.00	117.42	0.00	20	0.00	0.00	int maxVal<int>(TNT::Matrix<int> ...
0.00	117.42	0.00	20	0.00	0.30	mut(TNT::Matrix<unsigned int> con ...
0.00	117.42	0.00	20	0.00	0.00	TNT::Matrix<unsigned int>::newsiz ...
0.00	117.42	0.00	20	0.00	0.00	TNT::Matrix<double> partialSums<d ...
0.00	117.42	0.00	20	0.00	0.00	randScalar(void)
0.00	117.42	0.00	20	0.00	0.03	TNT::Matrix<int> rangeVect<int>(i ...
0.00	117.42	0.00	20	0.00	0.08	ranking(TNT::Matrix<double> const ...
0.00	117.42	0.00	20	0.00	1.08	recombin(RecombinFun, TNT::Matrix ...
0.00	117.42	0.00	20	0.00	0.12	reins(TNT::Matrix<unsigned int> c ...
0.00	117.42	0.00	20	0.00	0.18	select(ReprodSelFun, TNT::Matrix< ...
0.00	117.42	0.00	20	0.00	0.12	sus(TNT::Matrix<double> const &, ...
0.00	117.42	0.00	20	0.00	1.08	xovdp(TNT::Matrix<unsigned int> c ...
0.00	117.42	0.00	20	0.00	1.08	xovmp(TNT::Matrix<unsigned int> c ...
0.00	117.42	0.00	2	0.00	0.00	__static_initialization_and_destr ...
0.00	117.42	0.00	1	0.00	0.00	TNT::Matrix<unsigned int>::operat ...
0.00	117.42	0.00	1	0.00	0.01	TNT::Matrix<unsigned int> operato ...
0.00	117.42	0.00	1	0.00	1.15	istream & TNT::operator>><double> ...
0.00	117.42	0.00	1	0.00	0.00	changeFileExtension(char const *, ...
0.00	117.42	0.00	1	0.00	0.03	crtbp(unsigned int, unsigned int, ...
0.00	117.42	0.00	1	0.00	117417.65	ga_fs01(unsigned int, TNT::Matri ...
0.00	117.42	0.00	1	0.00	1.15	getInputs(void)
0.00	117.42	0.00	1	0.00	0.03	get_classcnt(TNT::Matrix<double> ...
0.00	117.42	0.00	1	0.00	117417.65	gracce_exop(void)
0.00	117.42	0.00	1	0.00	117417.65	gracce_fsel(void)
0.00	117.42	0.00	1	0.00	1.19	gracce_load(void)
0.00	117.42	0.00	1	0.00	1.19	load_ds(void)
0.00	117.42	0.00	1	0.00	0.00	double maxVal<double>(TNT::Matrix ...
0.00	117.42	0.00	1	0.00	0.00	int minPos<double>(TNT::Matrix<do ...
0.00	117.42	0.00	1	0.00	0.00	void printOnlyArray<unsigned int> ...

D GProf Call Graph of GRaCCE FS

The following is the call graph output from the GNU profiler *gprof* for a single run of GRaCCE.

The feature selection option was chosen and executed on the Iris data set. Lines in the output, shortened to prevent wrapping of the text, contain an ellipses marking the removed text.

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

index A unique number given to each element of the table.

Index numbers are sorted numerically.

The index number is printed next to every function name so it is easier to look up where the function in the table.

% time This is the percentage of the 'total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called.

If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a

member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

granularity: each sample hit covers 4 byte(s) for 0.01% of 117.42 seconds

index	% time	self	children	called	name
[1]	100.0	0.00	117.42		<spontaneous>
					main [1]
		0.00	117.42	1/1	gracce_exop(void) [2]
		0.00	0.00	1/1	gracce_load(void) [93]
[2]	100.0	0.00	0.00	1/1	getInputs(void) [96]
		0.00	117.42	1/1	
		0.00	117.42	1	main [1]
		0.00	117.42	1/1	gracce_exop(void) [2]
[3]	100.0	0.00	117.42	1/1	gracce_fsel(void) [3]
		0.00	117.42	1	gracce_exop(void) [2]
		0.00	117.42	1/1	gracce_fsel(void) [3]
		0.00	0.00	1/47090598	ga_fs01(unsigned int, TNT::Matrix<u...
		0.00	0.00	1/8580547	TNT::Matrix<double>::~~Matrix(void) [20]
		0.00	0.00	1/207	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	1/1	TNT::Matrix<unsigned int>::Matrix(...[400]
		0.00	0.00	1/1	changeFileExtension(char const *, ...[161]
[4]	100.0	0.00	0.00	1/1	void printOnlyArray<unsigned int>(...[162]
		0.00	0.00	1/3576	TNT::Matrix<unsigned int>::~~Matrix...[396]
		0.00	117.42	1/1	gracce_fsel(void) [3]
[4]	100.0	0.00	117.42	1	ga_fs01(unsigned int, TNT::Matrix<unsign...
		0.00	117.38	21/21	objfun2(unsigned int, TNT::Matrix<u...

		0.00	0.02	20/20	recombin(RecombinFun, TNT::Matrix<u...[56]
		0.00	0.01	20/20	mut(TNT::Matrix<unsigned int> const...[75]
		0.00	0.00	20/20	select(ReprodSelFun, TNT::Matrix<un...[79]
		0.00	0.00	20/20	reins(TNT::Matrix<unsigned int> cons...
		0.00	0.00	20/20	ranking(TNT::Matrix<double> const &)[88]
		0.00	0.00	63/423	TNT::Matrix<unsigned int>::operator...[110]
		0.00	0.00	1/1	crtbp(unsigned int, unsigned int, ...[126]
		0.00	0.00	1/1	get_classcnt(TNT::Matrix<double> c...[127]
		0.00	0.00	20/25567272	TNT::Matrix<double>::operator=(TNT...[10]
		0.00	0.00	1/121	TNT::Matrix<unsigned int> mat<unsig...[90]
		0.00	0.00	1/1	TNT::Matrix<unsigned int> operator...[131]
		0.00	0.00	20/21	double minVal<double>(TNT::Matrix<...[132]
		0.00	0.00	1/191	TNT::Matrix<double> appendCols<doub...[92]
		0.00	0.00	25/47090598	TNT::Matrix<double>::~~Matrix(void) [20]
		0.00	0.00	40/44119119	TNT::Matrix<double>::operator()(in...[28]
		0.00	0.00	1/1785	TNT::Matrix<unsigned int>::Matrix(i...[76]
		0.00	0.00	1/17131578	TNT::Matrix<double>::Matrix(int, i...[15]
		0.00	0.00	1/23	TNT::Matrix<double>::newsize(int, ...[129]
		0.00	0.00	1/1	int minPos<double>(TNT::Matrix<dou...[138]
		0.00	0.00	1/1761	TNT::Matrix<unsigned int>::operator...[108]
		0.00	0.00	2/13344733	TNT::Index1D::Index1D(int, int) [43]
		0.00	0.00	3/8580547	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	1/17133090	TNT::Matrix<double>::num_rows(void...[37]
		0.00	0.00	1/21378356	TNT::Matrix<double>::num_cols(void...[42]
		0.00	0.00	68/3576	TNT::Matrix<unsigned int>::~~Matrix...[396]
		0.00	0.00	3/207	TNT::Matrix<unsigned int>::Matrix(...[400]

[5]	100.0	0.00	117.38	21/21	ga_fs01(unsigned int, TNT::Matrix<un...[4]
		0.00	117.38	21	objfun2(unsigned int, TNT::Matrix<unsig...[4]
		0.11	117.25	190/190	knn2(unsigned int, TNT::Matrix<unsi...[4]
		0.00	0.02	190/190	get_featsubset(int, TNT::Matrix<uns...[64]
		0.00	0.00	380/380	TNT::Matrix<unsigned int> mat<unsig...[81]
		0.00	0.00	211/25567272	TNT::Matrix<double>::operator=(TNT...[10]
		0.00	0.00	380/920	TNT::Matrix<unsigned int>::operator...[118]
		0.00	0.00	21/4246521	TNT::Matrix<double> TNT::transpose<...[16]
		0.00	0.00	253/47090598	TNT::Matrix<double>::~~Matrix(void) [20]
		0.00	0.00	232/13344733	TNT::Index1D::Index1D(int, int) [43]
		0.00	0.00	21/23	TNT::Matrix<double>::newsize(int, ...[129]
		0.00	0.00	190/16994215	TNT::Matrix<double>::operator()(int) [38]
		0.00	0.00	190/4305073	TNT::Index1D::operator=(TNT::Index1...[48]
		0.00	0.00	42/8580547	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	380/3576	TNT::Matrix<unsigned int>::~~Matrix...[396]
		0.00	0.00	190/380	unsigned int sum<unsigned int>(TNT...[154]
		0.00	0.00	21/2343	TNT::Matrix<unsigned int>::num_row...[143]
		0.00	0.00	21/2323	TNT::Matrix<unsigned int>::num_col...[144]

[6]	99.9	0.11	117.25	190/190	objfun2(unsigned int, TNT::Matrix<unsi...[4]
		0.11	117.25	190	knn2(unsigned int, TNT::Matrix<unsigned in...[4]
		3.86	113.09	28500/28500	sortBydist2(bool, double, TNT::Matri...[7]
		0.06	0.09	28500/29026	TNT::Matrix<double> mat<double>(TNT...[54]
		0.00	0.02	28690/25567272	TNT::Matrix<double>::operator=(TNT...[10]
		0.02	0.00	28500/28500	int maxPos<int>(TNT::Matrix<int> co...[60]
		0.00	0.02	28500/28500	TNT::Matrix<int>::operator=(int con...[63]
		0.02	0.00	228570/44119119	TNT::Matrix<double>::operator()(in...[28]
		0.00	0.01	28500/8522216	TNT::Matrix<double>::operator()(TNT...[24]
		0.00	0.01	58520/47090598	TNT::Matrix<double>::~~Matrix(void) [20]
		0.01	0.00	256500/265047	TNT::Matrix<int>::operator()(int) [68]
		0.01	0.00	114000/45496310	TNT::Matrix<double>::operator()(in...[31]
		0.00	0.01	380/460	TNT::Matrix<int>::Matrix(int, int, ...[74]

		0.00	0.00	570/29125	TNT::Matrix<double> mat<double>(TNT... [53]
		0.00	0.00	29830/13344733	TNT::Index1D::Index1D(int, int) [43]
		0.00	0.00	29070/4305073	TNT::Index1D::operator=(TNT::Index1... [48]
		0.00	0.00	570/590	TNT::Matrix<double> operator/<doubl... [89]
		0.00	0.00	570/4333608	TNT::Region2D<TNT::Matrix<double> >... [22]
		0.00	0.00	1140/4704976	TNT::Matrix<double>::operator() (TNT... [32]
		0.00	0.00	190/17131578	TNT::Matrix<double>::Matrix(int, i... [15]
		0.00	0.00	190/17133090	TNT::Matrix<double>::num_rows(void... [37]
		0.00	0.00	190/21378356	TNT::Matrix<double>::num_cols(void... [42]
		0.00	0.00	190/8580547	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	1710/4120	TNT::Matrix<unsigned int>::operator... [394]
		0.00	0.00	380/640	TNT::Matrix<int>::~Matrix(void) [399]
		0.00	0.00	190/380	unsigned int sum<unsigned int>(TNT... [154]

[7]	99.6	3.86	113.09	28500/28500	knn2(unsigned int, TNT::Matrix<unsig...
		3.86	113.09	28500	sortBydist2(bool, double, TNT::Matrix<do... [7]
		0.88	24.32	28500/28501	VecQueue<double>::buildByRows(void) [9]
		3.57	13.50	4246500/4246500	TNT::Matrix<double> operator-<doubl... [13]
		0.27	15.74	4246500/4246500	TNT::Matrix<double> TNT::operator*... [16]
		3.18	10.35	4246500/4246521	TNT::Matrix<double> TNT::transpose<... [18]
		1.65	9.56	4246500/4246500	TNT::Matrix<double> map_sqrt<doub... [18]
		1.64	9.21	12796500/25567272	TNT::Matrix<double>::operator=(TN... [10]
		0.86	4.36	4246500/4247176	VecQueue<double>::enqueue(TNT::Matr... [23]
		0.52	4.41	8493000/8522216	TNT::Matrix<double>::operator() (TNT... [24]
		1.42	2.02	28500/28500	TNT::Matrix<double> sortNSmallestRo... [29]
		0.95	2.41	17128500/47090598	TNT::Matrix<double>::~Matrix(void) [20]
		1.24	0.00	16986000/16994215	TNT::Matrix<double>::operator() (int) [38]
		0.34	0.00	4389000/13344733	TNT::Index1D::Index1D(int, int) [43]
		0.31	0.00	4246500/45496310	TNT::Matrix<double>::operator() (in... [31]
		0.30	0.00	4275000/4305073	TNT::Index1D::operator=(TNT::Index1... [48]
		0.00	0.02	28500/21378323	TNT::Matrix<double>::Matrix(TNT::M... [11]
		0.00	0.02	28500/17131578	TNT::Matrix<double>::Matrix(int, i... [15]
		0.02	0.00	28500/28691	VecQueue<double>::VecQueue(void) [62]
		0.02	0.00	28500/28691	VecQueue<double>::~VecQueue(void) [61]
		0.00	0.00	57000/8580547	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	28500/17133090	TNT::Matrix<double>::num_rows(void... [37]
		0.00	0.00	28500/21378356	TNT::Matrix<double>::num_cols(void... [42]

		0.00	0.00	23/55449967	TNT::Matrix<double>::newsize(int,... [129]
		0.00	0.00	150/55449967	TNT::Matrix<double>::Matrix(int, ... [121]
		8.83	0.00	16939893/55449967	TNT::Matrix<double>::operator=(TN... [10]
		8.93	0.00	17131578/55449967	TNT::Matrix<double>::Matrix(int, ... [15]
		11.14	0.00	21378323/55449967	TNT::Matrix<double>::Matrix(TNT:... [11]
[8]	24.6	28.89	0.00	55449967	TNT::Matrix<double>::initialize(int, int) [8]

		0.00	0.00	1/28501	load_ds(void) [94]
		0.88	24.32	28500/28501	sortBydist2(bool, double, TNT::Matri... [7]
[9]	21.5	0.88	24.32	28501	VecQueue<double>::buildByRows(void) [9]
		0.89	9.64	4246650/4247176	VecQueue<double>::dequeue(void) [19]
		2.05	4.13	4246650/4333608	TNT::Region2D<TNT::Matrix<double> >... [22]
		0.54	3.06	4246650/25567272	TNT::Matrix<double>::operator=(TNT... [10]
		0.14	2.31	4246650/4704976	TNT::Matrix<double>::operator() (TNT... [32]
		0.24	0.60	4303652/47090598	TNT::Matrix<double>::~Matrix(void) [20]
		0.66	0.00	8493300/13344733	TNT::Index1D::Index1D(int, int) [43]
		0.00	0.02	28501/21378323	TNT::Matrix<double>::Matrix(TNT::M... [11]
		0.00	0.02	28501/17131578	TNT::Matrix<double>::Matrix(int, i... [15]
		0.00	0.01	28501/28691	VecQueue<double>::vecSize(void) const [67]
		0.00	0.00	28501/8580547	TNT::Matrix<double>::Matrix(void) [45]
		0.00	0.00	28501/28691	VecQueue<double>::queueSize(void) ... [139]

	0.00	0.00	1/25567272	load_ds(void) [94]
	0.00	0.00	2/25567272	crtbp(unsigned int, unsigned int,...[126]
	0.00	0.00	20/25567272	ga_fs01(unsigned int, TNT::Matrix<u...[4]
	0.00	0.00	40/25567272	ranking(TNT::Matrix<double> const &) [88]
	0.00	0.00	60/25567272	sus(TNT::Matrix<double> const &, u...[83]
	0.00	0.00	60/25567272	mut(TNT::Matrix<unsigned int> cons...[75]
	0.00	0.00	60/25567272	void sort<double>(TNT::Matrix<doub...[84]
	0.00	0.00	100/25567272	xovmp(TNT::Matrix<unsigned int> co...[58]
	0.00	0.00	211/25567272	objfun2(unsigned int, TNT::Matrix<uns...[80]
	0.00	0.00	526/25567272	VecQueue<double>::buildByCols(void) [80]
	0.00	0.02	28690/25567272	knn2(unsigned int, TNT::Matrix<unsign...[9]
	0.54	3.06	4246650/25567272	VecQueue<double>::buildByRows(void) [9]
	0.54	3.06	4247176/25567272	VecNode<double>::VecNode(TNT::Matr...[27]
	0.54	3.06	4247176/25567272	VecQueue<double>::dequeue(void) [19]
	1.64	9.21	12796500/25567272	sortbydist2(bool, double, TNT::Mat...[7]
[10]	18.5	3.28	18.40 25567272	TNT::Matrix<double>::operator=(TNT::Ma...[10]
		8.83	0.00 16939893/55449967	TNT::Matrix<double>::initialize(in...[8]
		7.19	0.00 25567272/46945595	TNT::Matrix<double>::copy(double ...[17]
		2.38	0.00 16939893/64030514	TNT::Matrix<double>::destroy(void) [21]

	0.00	0.00	20/21378323	ranking(TNT::Matrix<double> const &) [88]
	0.00	0.00	20/21378323	TNT::Matrix<double> partialSums<d...[122]
	0.00	0.00	20/21378323	TNT::Matrix<double> TNT::operator...[123]
	0.00	0.00	40/21378323	TNT::Matrix<double> rangeVect<dou...[120]

E Main GRaCCE Module

```
/* *****
/* File Name: gracce.cpp
/* Description: This file is the main driver for the various functions
/*      in the GRaCCE algorithm.
/* *****

/* include files removed for brevity */

//using namespace std;

/* *****
/*
/* MODULE NAME: gracce_exop
/* CALLED FROM: gracce_main (triggered by GUI event).
/* INPUT(S): None.
/* OUTPUT(S): None.
/* DESCRIPTION: This module initiates the operation selected by the
/* user.
/*
/* *****

void gracce_exop() {
switch (MainMenu::phaseChoice) {
case (WINNOWING):
gracce_winnow();
break;
case (FEATURE_SELECTION):
gracce_fsel();
break;
case (RULE_INDUCTION):
gracce_exec();
break;
} //switch
}

void main(int argc, char **argv) {

    MPI::Init(argc, argv); // DMS
    int comm_size = MPI::COMM_WORLD.Get_size(); // DMS
    int my_rank = MPI::COMM_WORLD.Get_rank(); // DMS

    char iLine[101];
    int bogomips = 0;
    ifstream inStream;
    int i;
    bool found;

    // open the cpu info file
    inStream.open("/proc/cpuinfo", ios::in);

    if (!inStream) {
```

```

cerr << "Could not open /proc/cpuinfo" << endl;
exit(0);
}

    found = false;
    while (!inStream.eof() & !found) {
inStream.getline(iLine, 100);
if ((iLine[0] == 'b') && (iLine[1] == 'o') && (iLine[2] == 'g')) {

    i = 10;
    while ((iLine[i] == ' ') || (iLine[i] == ':')) {
        i++;
    }
    // convert number into real
    while (iLine[i] != '.') {
        bogomips = bogomips * 10 + (int(iLine[i]) - 48);
        i++;
    }

    found = true;
}

        }

        inStream.close();

        MPI::COMM_WORLD.Gather(&bogomips, 1, MPI_INT, Shared::bogovec,
                                1, MPI_INT, 0);

        if (my_rank == 0) {

char dummyString[40];
int choiceInt;

ifstream inStream;

do {
cerr << "1) Winnowing Phase" << endl;
cerr << "2) Feature Selection" << endl;
cerr << "3) Rule Induction Phase" << endl;
cerr << "4) Exit" << endl << endl;
cerr << "Enter the number corresponding to your choice: ";
cin >> choiceInt;
MainMenu::phaseChoice = static_cast<Phase>(choiceInt-1);
cerr << endl;

switch (MainMenu::phaseChoice) {
case WINNOWING:
case FEATURE_SELECTION:
getInputs();
cerr << "Enter the file name (with extension) of the ";
cerr << "initial data set: ";
ws(cin);
cin.getline(MainMenu::FileName,50);
cerr << endl << endl;

```

```

cout << endl << "File Tested: " << MainMenu::FileName
    << endl << endl;
gracce_load();
if (MainMenu::phaseChoice == WINNOWING) {
cerr << "Press enter to begin Winnowing";
}
else {
cerr << "Press enter to begin Feature Selection";
}
cin.getline(dummyString,40);
break;
case RULE_INDUCTION:
getInputs();
cerr << "Enter the file name (with extension) of the ";
cerr << "winnowed data set: ";
ws(cin);
cin.getline(MainMenu::FileName,50);
cerr << endl << endl;
cout << endl << "File Tested: " << MainMenu::FileName
    << endl << endl;
gracce_load();
cerr << "Press enter to begin Rule Induction";
cin.getline(dummyString,40);
break;
case EXIT:
break;
default:
cerr << "Invalid selection" << endl;
break;
}
cerr << endl << endl;
gracce_exop();
cerr << endl << endl;
cout << endl << endl;
} while (MainMenu::phaseChoice != EXIT);

MPI::COMM_WORLD.Barrier();

    } // end if my_rank == 0
    else {
int buff_size = 64000;
real buffer[buff_size];
unsigned ubuffer[buff_size];
Subscript rt, rp;
Subscript ct, cp;
Matrix<real> wds; // winnowed data set after removing a feature
Matrix<real> cmx; // confusion matrix
real currMscr; // misclassification rate of current pop item
// Index1D rowI(1,1), allCols(1,cp);
Index1D rowI(1,1);
int i, j;

// ***** Get DataSets::AllData needed by knn *****
MPI::COMM_WORLD.Barrier(); // Barrier 1

```

```

MPI::COMM_WORLD.Bcast(&rt, 1, MPI::UNSIGNED, 0);
MPI::COMM_WORLD.Bcast(&ct, 1, MPI::UNSIGNED, 0);

for (i = 0; i < (rt * ct); i = i + buff_size) {
    if ((i+buff_size) < (rt * ct))
        MPI::COMM_WORLD.Bcast(&buffer[i], buff_size*2, MPI::REAL, 0);
    else
        MPI::COMM_WORLD.Bcast(&buffer[i], (rt*ct - i)*2, MPI::REAL, 0);
}

DataSets::AllData.newsize(rt,ct);

for (i = 1; i <= rt; i++) {
    for (j = 1; j <= ct; j++) {
        DataSets::AllData(i,j) = buffer[(j-1) + (i-1) * (int)ct];
    }
}

MPI::COMM_WORLD.Barrier(); // Barrier 2

// ***** get two variables required by knn2 function *****
unsigned sv[2];
MPI::COMM_WORLD.Bcast(sv, 2, MPI::UNSIGNED, 0);
Shared::ClassMax = sv[0];
Shared::ClassMin = sv[1];

unsigned finished = 0; // Flag used to signal root has finished

// ***** objfun2 receives the following five items *****
unsigned vok;
Matrix<unsigned> tcc;
Matrix<unsigned> pop;
real mscr;
Matrix<real> score;

int rp_start, rp_end; // boundaries for modified for loop
unsigned Assignments[1024]; // holds assignments for 1024 / 4 nodes

while (finished == 0) { // loop until root sends a finished Bcast
    MPI::COMM_WORLD.Bcast(&finished, 1, MPI::UNSIGNED, 0);
    if (finished == 0) {

        // ***** get vok *****
        MPI::COMM_WORLD.Bcast(&vok, 1, MPI::UNSIGNED, 0);

        // ***** get tcc *****
        MPI::COMM_WORLD.Bcast(&rt, 1, MPI::UNSIGNED, 0);
        MPI::COMM_WORLD.Bcast(&ct, 1, MPI::UNSIGNED, 0);
        tcc.newsize(rt,ct);

        for (i = 0; i < (rt * ct); i = i + buff_size) {
            if ((i+buff_size) < (rt * ct))
                MPI::COMM_WORLD.Bcast(&ubuffer[i], buff_size, MPI::UNSIGNED, 0);
            else

```

```

MPI::COMM_WORLD.Bcast(&ubuffer[i], (rt*ct - i), MPI::UNSIGNED, 0);
}

for (i = 1; i <= rt; i++) {
    for (j = 1; j <= ct; j++) {
tcc(i,j) = ubuffer[(j-1) + (i-1) * (int)ct];
    }
}

// ***** get pop *****
MPI::COMM_WORLD.Bcast(&rp, 1, MPI::UNSIGNED, 0);
MPI::COMM_WORLD.Bcast(&cp, 1, MPI::UNSIGNED, 0);
pop.newsize(rp,cp);

for (i = 0; i < (rp * cp); i = i + buff_size) {
    if ((i+buff_size) < (rp * cp))
MPI::COMM_WORLD.Bcast(&ubuffer[i], buff_size, MPI::UNSIGNED, 0);
    else
MPI::COMM_WORLD.Bcast(&ubuffer[i], (rp*cp - i), MPI::UNSIGNED, 0);
}

for (i = 1; i <= rp; i++) {
    for (j = 1; j <= cp; j++) {
pop(i,j) = ubuffer[(j-1) + (i-1) * (int)cp];
    }
}

Index1D allCols(1,cp); // create index based on population size

score.newsize(1,rp);
mscr = Constants::INFINITY;
currMscr = mscr;
rp_start = 0;
rp_end = 0;

// For each individual, reconstruct and evaluate a subset of
// TrainData using the chosen features.

int buflength = comm_size * 4;
MPI::COMM_WORLD.Barrier();
MPI::COMM_WORLD.Bcast(Assignments, buflength, MPI_INT, 0);

for (int j = 0; j < comm_size; j++) {
    if (Assignments[j*4] == my_rank) {
rp_start = Assignments[(j*4)+1];
rp_end = rp_start + Assignments[(j*4)+2] - 1;
    }
}

MPI::COMM_WORLD.Barrier();

for (i = rp_start; i <= rp_end; i++) {
    rowI = Index1D(i,i);

```

```

        // Only test fmaps that include at least 1 feature
        if (sum(mat(pop(rowI,allCols))) > 0) {
wds = get_featsubset(1,mat(pop(rowI,allCols)),
        DataSets::AllData);
knn2(vok,tcc,wds,currMscr,cmx,score(i));
if (currMscr < mscr) {
    mscr = currMscr;
}
        }
        else {
score(i) = 1.0;
        } // else
    } // for

    // return lowest mscr to root node
    MPI::COMM_WORLD.Reduce(&mscr, &currMscr, 1, MPI::REAL, MPI::MIN, 0);

    // send parts of score back to root node
    MPI::COMM_WORLD.Send(&rp_start, 1, MPI::INT, 0, 0);
    MPI::COMM_WORLD.Send(&rp_end, 1, MPI::INT, 0, 0);

    for (i = rp_start; i <= rp_end; i++)
        buffer[i] = score(i);

    if (rp_end >= rp_start)
        MPI::COMM_WORLD.Send(&buffer[rp_start], (rp_end - rp_start + 1),
            MPI::DOUBLE, 0, 0);
    else
        MPI::COMM_WORLD.Send(&dummy, 1, MPI::DOUBLE, 0, 0);
}
}

// wait for end of root node
MPI::COMM_WORLD.Barrier();
}

    MPI::Finalize();
}

```


F GRaCCE's Feature Selection Module

```

/*****
/* File Name: featSel.cpp
/* Description: This file contains functions used during the feature
/*      selection phase of the GRaCCE algorithm.
/* Public Functions:
/*      gracce_fsel
/* Private Functions:
/*      knn2
/*      objfun2
/*      ga_fs01
/*      ga_fs02
*****/

/* Include files deleted for brevity */

#define HETERO
using namespace std;

/*****
/*
/* MODULE NAME: knn2
/* CALLED FROM: objfun2
/* INPUT(S):
/* K - Value of K (for the kNN procedure).
/* tcc - Per class tally (of instances).
/* ds - Data set to be processed.
/*
/* OUTPUT(S):
/* mscr - Overall misclassification rate.
/* cmx - Confusion matrix (based on the kNN results) for the
/*      remaining data. Each (i,j) element is the percentage of the
/*      data points in ds of class i that have K-closest neighbors of
/*      class j.
/* score - Overall fitness (accounts for number of features used
/*      in the data set.
/*
/* DESCRIPTION: This module uses K-Nearest Neighbor algorithm to classify
/* each data point (based on its K closest neighbors). The
/* results are computed and the score (using the input)
/* data set is returned. This is a no-frills kNN
/* implementation used in conjunction with the feature
/* selection function.
/*
*****/

void knn2(unsigned K, const Matrix<unsigned> &tcc,
          const Matrix<real> &ds,
          real &mscr, Matrix<real> &cmx, real &score) {

unsigned nc = (Shared::ClassMax-Shared::ClassMin)+1; // num classes
int dc = 1-Shared::ClassMin; // class offset into arrays

```

```

Subscript rows = ds.num_rows();
Subscript cols = ds.num_cols();

Matrix<int> ccnt(nc,1); // count of instances of each class
Subscript indx;        // position in ccnt with highest count
Matrix<real> nn_list;    // Sorted list of points, where:
                        // nn_list(i,:) = [class distance index]
Matrix<Subscript> map(rows,1,0); // class mapping for each exemplar

Index1D rowJ(1,1), afterCol1(2,cols);
Subscript i,j,c;

for (j=1; j<=rows; j++) {

rowJ = Index1D(j,j);

nn_list = sortbydist2(false,ds(j,1),mat(ds(rowJ,afterCol1)),
                        j,ds,K);

// Determine the majority class and update the class mapping.
ccnt = 0; // init ALL elements in ccnt to 0

for (i=1; i<=K; i++)
ccnt((int)nn_list(i,1)+dc) = ccnt((int)nn_list(i,1)+dc) + 1;

indx = maxPos(ccnt);
map(j) = indx-dc;

} //for j

// Now compute the confusion matrix for the entire data set.
cmx = Matrix<real>(nc,nc,0.0);

for (c=1; c<=nc; c++) {
for (i=1; i<=rows; i++) {
if (static_cast<Subscript>(ds(i,1)) == (c-dc)) {
cmx(c,map(i)+dc) = cmx(c,map(i)+dc)+1.0;
} //if
} //for i
} //for c

Index1D rowC(1,1), allCols(1,nc);
real res; // percentage of pts of given class that are misclassified

// Convert confusion matrix to percentages and compute
// the average per class mis-classification rate.
mscr = 0.0;
for (c=1; c<=nc; c++) {
if (tcc(c) > 0) {
rowC = Index1D(c,c);
cmx(rowC,allCols) = mat(cmx(rowC,allCols))
/ static_cast<real>(tcc(c));

```

```

    res = 1.0 - cmx(c,c);
    mscr += res * static_cast<real>(tcc(c));
} //if
} //for c

mscr = mscr/static_cast<real>(sum(tcc)); // Overall error rate.
score = mscr+(0.005*static_cast<real>((cols-1))); // Overall fitness.
}

/*****
/*
/* MODULE NAME: objfun2
/* CALLED FROM: ga_fs01,
/*             ga_fs02
/* INPUT(S):
/* vok - Value of K (for kNN algorithm).
/* tcc - Tally of instances per target class.
/* pop - Population of solutions (to be evaluated).
/*
/* OUTPUT(S):
/* mscr - Best overall misclassification rate.
/* score - Array of fitness scores (for each individual).
/*
/* DESCRIPTION: This objective function evaluates the performance of
/* the selected feature set (for a given data set) on
/* the kNN algorithm.
/*
*****/

void objfun2(unsigned vok, const Matrix<unsigned> &tcc,
             const Matrix<unsigned> &pop,
             real &mscr, Matrix<real> &score) {
    Subscript rp = pop.num_rows();
    Subscript cp = pop.num_cols();

    int comm_size = MPI::COMM_WORLD.Get_size();
    int my_rank = MPI::COMM_WORLD.Get_rank();

    int buff_size = 64000;
    unsigned ubuffer[buff_size];
    real buffer[buff_size];
    real dummy = -1.0;
    int rp_start, rp_end;
        unsigned Assignments[1024];

    unsigned finished = 0;
    MPI::COMM_WORLD.Bcast(&finished, 1, MPI::UNSIGNED, 0);

    Matrix<real> wds; // winnowed data set after removing a feature
    Matrix<real> cmx; // confusion matrix
    real currMscr; // misclassification rate of current pop item

```

```

Index1D rowI(1,1), allCols(1,cp);
Subscript i, j;
Subscript rt, ct;

score.newsize(1,rp);
mscr = Constants::INFINITY;

// ***** sending vok *****
MPI::COMM_WORLD.Bcast(&vok, 1, MPI::UNSIGNED, 0);

// ***** sending tcc *****
rt = tcc.num_rows();
ct = tcc.num_cols();
MPI::COMM_WORLD.Bcast(&rt, 1, MPI::UNSIGNED, 0);
MPI::COMM_WORLD.Bcast(&ct, 1, MPI::UNSIGNED, 0);

for (i = 1; i <= rt; i++) {
    for (j = 1; j <= ct; j++) {
        ubuffer[(j-1) + (i-1) * (int)ct] = tcc(i,j);
    }
}

for (i = 0; i < (rt * ct); i = i + buff_size) {
    if ((i+buff_size) < (rt * ct))
        MPI::COMM_WORLD.Bcast(&ubuffer[i], buff_size, MPI::UNSIGNED, 0);
    else
        MPI::COMM_WORLD.Bcast(&ubuffer[i], (rt*ct - i), MPI::UNSIGNED, 0);
}

// ***** sending pop *****
MPI::COMM_WORLD.Bcast(&rp, 1, MPI::UNSIGNED, 0);
MPI::COMM_WORLD.Bcast(&cp, 1, MPI::UNSIGNED, 0);

for (i = 1; i <= rp; i++) {
    for (j = 1; j <= cp; j++) {
        ubuffer[(j-1) + (i-1) * (int)cp] = pop(i,j);
    }
}

for (int i = 0; i < (rp * cp); i = i + buff_size) {
    if ((i+buff_size) < (rp * cp))
        MPI::COMM_WORLD.Bcast(&ubuffer[i], buff_size, MPI::UNSIGNED, 0);
    else
        MPI::COMM_WORLD.Bcast(&ubuffer[i], (rp*cp - i), MPI::UNSIGNED, 0);
}

// For each individual, reconstruct and evaluate a subset of
// TrainData using the chosen features.

// Initialize Assignments vector
for (int j = 0; j < comm_size; j++) {
    Assignments[j*4] = j;          // node number
    Assignments[j*4 + 1] = 0;      // population member to start on
    Assignments[j*4 + 2] = 0;      // number of members to process
}

```

```

    Assignments[j*4 + 3] = (unsigned)Shared::bogovec[j];
}

#ifdef HETERO // adds a simple bubble sort on Assignments vector
bool changed = true;
while (changed) {
    changed = false;
    for (int j = 0; j < comm_size - 1; j++)
        if (Assignments[j*4 + 3] < Assignments[(j+1)*4 + 3]) {
            Assignments[comm_size*4 + 0] = Assignments[j*4 + 0];
            Assignments[comm_size*4 + 1] = Assignments[j*4 + 1];
            Assignments[comm_size*4 + 2] = Assignments[j*4 + 2];
            Assignments[comm_size*4 + 3] = Assignments[j*4 + 3];
            Assignments[j*4 + 0] = Assignments[(j+1)*4 + 0];
            Assignments[j*4 + 1] = Assignments[(j+1)*4 + 1];
            Assignments[j*4 + 2] = Assignments[(j+1)*4 + 2];
            Assignments[j*4 + 3] = Assignments[(j+1)*4 + 3];
            Assignments[(j+1)*4 + 0] = Assignments[comm_size*4 + 0];
            Assignments[(j+1)*4 + 1] = Assignments[comm_size*4 + 1];
            Assignments[(j+1)*4 + 2] = Assignments[comm_size*4 + 2];
            Assignments[(j+1)*4 + 3] = Assignments[comm_size*4 + 3];
            changed = true;
        }
}
#endif

int place = 0;
for (int j = 0; j < rp; j++) {
    Assignments[(place*4)+2]++;
    place++;
    if (place == comm_size)
        place = 0;
}

Assignments[1] = 1;
for (int j = 0; j < comm_size - 1; j++) {
    Assignments[(j+1)*4 + 1] = Assignments[j*4 + 1] + Assignments[j*4 + 2];
}

int buflength = comm_size * 4;
MPI::COMM_WORLD.Barrier();
MPI::COMM_WORLD.Bcast(Assignments, buflength, MPI_INT, 0);

for (int j = 0; j < comm_size; j++) {
    if (Assignments[j*4] == my_rank) {
        rp_start = Assignments[(j*4)+1];
        rp_end = rp_start + Assignments[(j*4)+2] - 1;
    }
}

MPI::COMM_WORLD.Barrier();

for (i = rp_start; i <= rp_end; i++) {
    rowI = Index1D(i,i);
}

```

```

// Only test fmaps that include at least 1 feature
if (sum(mat(pop(rowI,allCols))) > 0) {
wds = get_featsubset(1,mat(pop(rowI,allCols)),
                    DataSets::AllData);
knn2(vok,tcc,wds,currMscr,cmx,score(i));
if (currMscr < mscr) {
    mscr = currMscr;
}
}
else {
score(i) = 1.0;
} //if
buffer[i] = score(i);
} //i

MPI::COMM_WORLD.Barrier();
// return lowest mscr to root node
MPI::COMM_WORLD.Reduce(&mscr, &currMscr, 1, MPI::REAL, MPI::MIN, 0);
if (currMscr < mscr)
    mscr = currMscr;

for (int j = 1; j < comm_size; j++) {
    // send parts of score back to root node
    MPI::COMM_WORLD.Recv(&rp_start, 1, MPI::INT, j, 0);
    MPI::COMM_WORLD.Recv(&rp_end, 1, MPI::INT, j, 0);
    if (rp_end >= rp_start)
        MPI::COMM_WORLD.Recv(&buffer[rp_start], (rp_end - rp_start + 1),
                            MPI::DOUBLE, j, 0);
    else
        MPI::COMM_WORLD.Recv(&dummy, 1, MPI::DOUBLE, j, 0);
}

for (i = 1; i <= rp; i++)
    score(i) = buffer[i];

// cout << "Score = " << score << endl;

score = transpose(score);
}

/*****
/*
/* MODULE NAME: ga_fs01
/* CALLED FROM: gracce_fsel
/* INPUT(S):
/* vok - Value of k for the kNN algorithm.
/*
/* OUTPUT(S):
/* fmap - List of enabled features in the best solution.
/* hist - (Best) Fitness history of the GA population, where for each
/*         row, col 1 = generation number, col 2 = fitness value.
/*
/* DESCRIPTION: This module performs feature selection, using a GA-based

```

```

/* search to find the best (most fit) feature set. Fitness
/* is based on the size of the feature set and its
/* classification accuracy achieved using the kNN algorithm.
/*
/*****

void ga_fs01(unsigned vok, Matrix<unsigned> &fmap, Matrix<real> &hist) {
Subscript rt = DataSets::AllData.num_rows();
Subscript ct = DataSets::AllData.num_cols();
Matrix<unsigned> tcc;          // class count for

const unsigned BASE = 2;      // Configure as a binary GA.
const unsigned LIND = ct-1;   // Set the chromosome size.
const unsigned MAXGEN = 20;   // Set the maximum number of generations.
unsigned NIND = Shared::PopSize; // Determine the population size.
cout << "Pop size = " << NIND - 1 << endl;

Matrix<unsigned> Chrom; // set of binary chromosomes where each row
                        // corresponds to a feature mapping
Matrix<unsigned> base = BASE*Matrix<unsigned>(1,LIND,1);

real er;              // best overall misclass. rate in a given population
Matrix<real> ObjV;     // objective vals for each row item in population
unsigned gen;         // current generation number
real mv;              // min objective value in a given population
Subscript mi;         // index of min objective value
Matrix<real> FitnV;     // fitness values for all objective values
Matrix<unsigned> SelCh; // chromosomes selected for recombination
Matrix<real> ObjVSel;   // objective vals for items in SelCh

// Tally the number of instances in each class.
tcc =
    get_classcnt(appendCols(DataSets::AllData,Matrix<real>(rt,1,0.0)));

// Initialize the population
Chrom = crtbp(NIND,LIND,BASE);
objfun2(vok,tcc,Chrom,er,ObjV);

hist.newsize(MAXGEN,2);

// Start the Generational loop
gen = 1;
while (gen <= MAXGEN) {
    cout << "gen = " << gen << endl;

    // Assign fitness
    // DMS FitnV = ranking(ObjV);
    FitnV = ranking(ObjV);

    // Select individuals for breeding
    SelCh = select(GA_Menu::SELF, Chrom, FitnV, GA_Menu::GGAP);

    // Recombine individuals (crossover)
    SelCh = recomb(GA_Menu::RECF, SelCh,GA_Menu::PCROS);

```

```

// Apply mutation
SelCh = mut(SelCh,GA_Menu::PMUT,base);

// Evaluate offspring, call objective function
objfun2(vok,tcc,SelCh,er,ObjVSel);

// Reinsert offspring into population
reins(SelCh,ObjVSel,Chrom,ObjV);

// Minimize the average (per class) misclassification rate
mv = minVal(ObjV);

hist(gen,1) = static_cast<real>(gen);
hist(gen,2) = mv;
gen++;
} //while

// Return the best feature set found.
mi = minPos(ObjV);
fmap = mat(Chrom(Index1D(mi,mi),Index1D(1,LIND)));
}

/*****
/*
/* MODULE NAME: ga_fs02
/* CALLED FROM: gracce_fsel
/* INPUT(S):
/* method - Determines if a forward search or hybrid search
/*           (limited fwd search/GA-based search) is utilized.
/*           If the hybrid search is chosen, a transition to a
/*           GA-based search is made after a limited fwd search.
/* maxext - Determines the percentage of features sought in the
/*           limited forward search.
/* vok      - Value of k for the kNN algorithm.
/*
/* OUTPUT(S):
/* fmap     - List of enabled features in the best solution.
/* hist     - (Best) Fitness history of the GA population, where for each
/*            row, col 1 = generation number, col 2 = fitness value.
/*
/* DESCRIPTION: This module performs feature selection by using either
/* a full deterministic forward search or a hybrid (limited
/* fwd search/GA-based search) approach. The limited fwd
/* search finds a feature set with which to dope the
/* population for the subsequent GA-based search. Feature
/* set fitness is based on the size of the feature set and
/* its classification accuracy achieved using the kNN
/* algorithm.
/*
/* *****/

/***** Code not modified - removed for brevity *****/

```



```

/*****
/*
/* MODULE NAME: gracce_fsel
/* CALLED FROM: gracce_exop
/* INPUT(S): None.
/* OUTPUT(S): None.
/* DESCRIPTION: This module performs the kNN based feature selection
/* process.
/*
/*****/

void gracce_fsel(){

    // PREPROCESSING (1): FEATURE SELECTION (if enabled)

    clock_t start, end;
    int buff_size = 64000;
    // send DataSets::AllData here
    Subscript rt = DataSets::AllData.num_rows();
    Subscript ct = DataSets::AllData.num_cols();
    real buffer[buff_size];
    int i, j;

    for (i = 1; i <= rt; i++) {
        for (j = 1; j <= ct; j++) {
            buffer[(j-1) + ((i-1) * ct)] = DataSets::AllData(i,j);
        }
    }

    MPI::COMM_WORLD.Barrier(); // Barrier 1
    MPI::COMM_WORLD.Bcast(&rt, 1, MPI::UNSIGNED, 0);
    MPI::COMM_WORLD.Bcast(&ct, 1, MPI::UNSIGNED, 0);

    for (i = 0; i < (rt * ct); i = i + buff_size) {
        if ((i+buff_size) < (rt * ct))
            MPI::COMM_WORLD.Bcast(&buffer[i], buff_size*2, MPI::REAL, 0);
        else
            MPI::COMM_WORLD.Bcast(&buffer[i], (rt*ct - i)*2, MPI::REAL, 0);
    }

    MPI::COMM_WORLD.Barrier(); // Barrier 2
    unsigned sv[2];
    sv[0] = Shared::ClassMax;
    sv[1] = Shared::ClassMin;
    MPI::COMM_WORLD.Bcast(sv, 2, MPI::UNSIGNED, 0);

    Matrix<unsigned> fmap; // feature mapping, where 1 = feature is
                        // enabled, 0 = feature is disabled
    Matrix<real> history; // history of

    // loop code here for testing purposes only
    for (int poploop = 1; poploop <= 20; poploop++) {
        Shared::PopSize = poploop+1;

```

```

for (int loop = 0; loop < 5; loop++) {
    cout << "Starting Loop " << loop << endl;
    start = clock();

    // Determine if a data set has been loaded.
    if (Shared::DataSetLoaded == 0) {
        cerr << "No data set is loaded" << endl;
        return;
    }
    // Initiate the proper feature selection approach.

    if (PreprocMenu::fsmth == GA_SRCH) { // GA-based feature selection.
        ga_fs01(PreprocMenu::vok,fmap,history);
    }
    else { // Forward search feature selection.
        ga_fs02(PreprocMenu::fsmth,PreprocMenu::maxext,
            PreprocMenu::vok,fmap,history);
    } //if

    // Now output the winnowed data set
    ofstream outStream;
    char fmapFileName[31];

    // cout << "Writing out winnowed data set." << endl; // DMS

    changeFileExtension(MainMenu::FileName, ".fsr", fmapFileName);
    outStream.open(fmapFileName, ios::out);
    printOnlyArray(outStream,fmap);

    end = clock();

    cout << "Time for Feature Selection: "
        << (static_cast<real>(end-start) / CLOCKS_PER_SEC) << "s"
    << endl;

    } // end loop code here
}
int finished = 1;

// tell other processes to exit loop
MPI::COMM_WORLD.Bcast(&finished, 1, MPI::INT, 0);
}

```

G GRaCCE Default Inputs

The following is the content of the default input file, *inputs.txt*, for GRaCCE. These settings were used for all experiments in this thesis effort.

*** Main Menu ***

Data Profile (1=50/50,2=60/40,3=75/25,4=80/20,5=90/10,6=CVAL-5,7=CVAL-10): 6
Test Fold (1-5 for Data Profile=CVAL-5,1-10 for VAL-10): 1

*** Preprocessing Menu ***

Feature Set Type (0=Full,1=Partial): 0
Value of K (1,2,3,4,5,10,20): 3
Feature Selection Method (0=GA SRCH, 1=FWD SRCH, 2=FWD-GA): 0
Forward Search Extent (any increment of 0.05 from 0.0 to 1.0): 0.2

*** Genetic Algorithms Menu ***

Probability of Crossover (any increment of 0.05 from 0.0 to 1.0): 0.70
Probability of Mutation (0.001, 0.005, 0.01, 0.025, 0.05, 0.075, 0.1): 0.1
Reproduction Selection Function (0=sus, 1=rws): 0
Generation Window Size (3,5,10,15,20,25,30,40,50): 10
Recombin. Function (0=xovsp,1=xovdp,2=xovdprs,3=xovmp,4=xovsh,5=xovshrs,
6=xovsprs): 1
Replacement Percentage (0.5, 0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0): 0.9
Population Size (50,75,100,200,300,500,750,1000): 100

*** Region Identification Menu ***

Percentage of Part. Utilization (any increment of 0.05 from 0.0 to 1.0): 1.0
Region Utility Ratio (0.0, 0.001, 0.005, 0.01, 0.02, 0.03, 0.05, 0.1,
0.2, 0.3): 0.01
Partition Simplification Ratio (any increment of 0.05 from 0.0 to 1.0): 0.10
Sample Size for Covariance Estimate(0,3,5,10,20, where 0=Automatic): 5
Degree of Purity for Regions (any increment of 0.05 from 0.0 to 1.0): 0.80
Partition Simplification Error Threshold (any increment of 0.05 from
0.0 to 1.0): 0.50
Partition Usage (0=Mixed,1=Global Only,2=Local Only): 0
Partition Simplification Method (0=None,1=Training,2=Winnowed,
3=Bpts,4=Wt. Bpts): 4

References

- [1] Paul S. Barth. Using and Managing the Data Warehouse, chapter Mining for Profits in the Data Warehouse, pages 137–149. Prentice Hall, 1997.
- [2] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. Evolutionary Computation 1. Institute of Physics Publishing, 2000.
- [3] T. Bäck, D. B. Fogel, and T. Michalewicz, editors. Evolutionary Computation 2. Institute of Physics Publishing, 2000.
- [4] R. E. Bellman. Adaptive Control Processes: A Guided Tour. Princeton University Press, 1961.
- [5] Christopher M. Bishop. Neural Networks for Pattern Recognition. Oxford University Press, 1984.
- [6] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.
- [7] Rajkumar Buyya. High Performance Cluster Computing, volume 1. Prentice-Hall, Inc., 1999.
- [8] Harlan Crowder, Ron S. Dembo, and John M. Mulvey. On reporting computational experiments with mathematical software. ACM Transactions on Mathematical Software, 1979.
- [9] R. A. Fischer. The use of multiple measurements in taxonomic problems. Annual Eugenics, 7(II):179–188, 1936.
- [10] Message Passing Interface Forum. Mpi: A message-passing interface standard, June 1995.
- [11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.
- [12] R. L. Grossman. Data mining: Challenges and opportunities for data mining during the next decade, technical report. Technical report, Laboratory for Advanced Computing, University of Illinois at Chicago, 1997.

- [13] John L. Gustafson and Quinn O. Snell. Hint: A new way to measure computer performance.
<http://www.scl.ameslab.gov/Publications/HINT/ComputerPerformance.html>.
- [14] Lonnie P. Hammack. Parallel data mining with the message passing interface standard on clusters of personal computers. Master's thesis, Air Force Institute of Technology, 1999.
- [15] Richard H. F. Jackson, Paul T. Boggs, Stephen G. Nash, and Susan Powell. Guidelines for reporting results of computational experiments. report of the ad hoc committee. In Mathematical Programming, 1991.
- [16] Christopher W. Kinzig. An efficient data mining system based on genetic algorithms. Master's thesis, Wright State University, 2000.
- [17] Robert O. Kuehl. Design of Experiments: Statistical Principles of Research Design and Analysis. Duxbury Press, 2000.
- [18] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. Introduction to Parallel Computing. The Benjamin Cummings Publishing Company, Inc., 1994.
- [19] Robert E. Marmelstein and Gary B. Lamont. A new approach for evolving clusters. ACM Symposium on Applied Computing, 1999.
- [20] Robert Evan Marmelstein. Evolving Compact Decision Rule Sets. PhD thesis, Air Force Institute of Technology, 1999.
- [21] Mathematical, National Institute of Standards Computational Sciences Division, and MD Technology, Gaithersburg. Template numerical toolkit. zip file.
- [22] David H. M.Spector. Building Linux Clusters. O'Reilly, 2000.
- [23] Michael J. Quinn. Parallel Computing, Theory and Practice. McGraw-Hill, Inc., 1994.
- [24] Wojciech Siedlecki and Jack Sklansky. On automatic feature selection. International Journal of Pattern Recognition and Artificial Intelligence, 2(2):197-220, 1988.

- [25] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI - The Complete Reference, volume Volume 1, The MPI Core. The Massachusetts Institute of Technology Press, second edition edition, 2000.
- [26] Sorting algorithms. <http://www.cs.rit.edu/~atk/Java/Sorting/sorting.html>.
- [27] General Gordon R. Sullivan and Lt Col James B. Dubik. War in the information age. Military Review, April 1994.
- [28] Ronald E. Walpole, Raymond H. Myers, and Sharon L. Myers. Probability and Statistics for Engineers and Scientists. Prentice Hall, 6th edition, 1998.
- [29] Matt Welsh, Matthias Kalle Dalheimer, and Lar Kaufman. Running Linux. O'Reilly, third edition, 1999.
- [30] Allan C. Wilson. The molecular basis of evolution. Scientific American, 1985.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</p>					
1. REPORT DATE/DD-MM-YYYY 20-03-2001		2. REPORT TYPE Master's Thesis		3. DATES COVERED/From - To/ November 2000 - March 2001	
4. TITLE AND SUBTITLE IMPLEMENTATION AND ANALYSIS OF THE PARALLEL GENETIC RULE AND CLASSIFIER CONSTRUCTION ENVIRONMENT				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
				5d. PROJECT NUMBER	
6. AUTHOR(S) Strong, David M.				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/01M-14	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Richard W. Linderman AFRL/IFTC, BLDG 106 Griffiss Technology & Business Park 32 Hanger Rd Rome, NY 13441-4114 (315) 330-2208				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED					
13. SUPPLEMENTARY NOTES Dr. Gary B. Lamont, Professor, (937) 255-3636 x4718, Gary.Lamont@afit.edu					
14. ABSTRACT This paper discusses the Genetic Rule and Classifier Construction Environment (GRaCCE), which is an alternative to existing decision rule induction (DRI) algorithms. GRaCCE is a multi-phase algorithm which uses evolutionary search to mine classification rules from data. The current implementation uses a genetic algorithm based 0/1 search to reduce the number of features to a minimal set of features that make the most significant contributions to the classification of the input data set. This feature selection increases the efficiency of the rule induction algorithm that follows. However, feature selection is shown to account for more than 98 percent of the total execution time of GRaCCE on the tested data sets. The primary objective of this research effort is to improve the overall performance of GRaCCE through the application of parallel computing methods to the feature selection algorithm. The development and implementation of a parallel feature selection algorithm is presented. The experiments designed and used to test this parallel implementation are outlined followed by an analysis of the results. The results of this thesis effort show clearly that GRaCCE is improved through the use of parallel programming techniques.					
15. SUBJECT TERMS genetic algorithm, feature selection, parallel computing					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Gary Lamont
U	U	U	UU	87	19b. TELEPHONE NUMBER/Include area code/ (937)255-3636 x 4718